Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Aleksander Lukić

Komunikacija v dvoprocesorskem sistemu s skupnim pomnilnikom

diplomsko delo
na univerzitetnem študiju

Mentor: dr. Andrej Brodnik

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V dvoprocesorskih sistemih je skupni pomnilnik običajni skupni in deljeni vir med procesorjema. Zato ga zato lahko uporabimo za usklajevanje in komunikacijo med procesorjema. V nalogi preučite različne možne načine za usklajevanje procesov, ki se izvajajo na ločenih procesorjih, ter kako lahko s pomočjo usklajevalnih mehanizmov procesorja komunicirata. Na podlagi preučenega gradiva se odločite za enega od mehanizmov za usklajevanje in komunikacijo ter ga implementirajte na asimetričnem dvojedrnem mikrokrmilniku LPC4350. Vaša implementacija naj sledi standardu POSIX. Implementacijo ovrednotite z rešitvijo konkretnega problema.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Aleksander Lukić, z vpisno številko **63010089**, sem avtor diplomskega dela z naslovom:

Komunikacija v dvoprocesorskem sistemu s skupnim pomnilnikom

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 10. marca 2015 Podpis avtorja:

Kazalo

Seznam uporabljenih kratic in simbolov

ADC	*(angl.)*	Analogue-to-Digital Converter; analogno-digitalni pretvornik
AHB	*(angl.)*	Advanced High-performance Bus; napredno visoko performančno vodilo
FIFO	*(angl.)*	First-In-First-Out; prvi noter, prvi ven
ICD	*(angl.)*	Implantable Cardioverter-Defibrilator; vsadni kardioverter-defibrilator
LED	*(angl.)*	Light-Emitting Diode; svetleča dioda
LIFO	*(angl.)*	Last-In-First-Out; zadnji noter, prvi ven
MOSFET	*(angl.)*	Metal–Oxide–Semiconductor Field-Effect Transistor; polprevodniški poljski tranzistor s kovinskim oksidom
MPU	*(angl.)*	Memory Protection Unit; enota za zaščito pomnilnika
NVIC	*(angl.)*	Nested Vector Interrupt Controller; vgnezden vektorski prekinitveni krmilnik
SRAM	*(angl.)*	Static Random-Access Memory; statični pomnilnik z naključnim dostopom
UART	*(angl.)*	Universal asynchronous Receiver-Transmitter; univerzalni asinhroni sprejemnik in oddajnik

Povzetek

Potreba po vedno zmogljivejših računalniških sistemih na eni strani in fizikalne omejitve na drugi so pripeljale do razvoja večjedrnih procesorjev, ki čedalje bolj prodirajo tudi v področje vgrajenih sistemov in neredko opravljajo različne časovno kritične funkcije. Na tovrstnih sistemih je posebej pomemben mehanizem komunikacije med različnimi programskimi opravili, ki se med sabo morajo usklajevati in si izmenjevati informacije na zanesljiv in časovno predvidljiv način. Posebna vrsta večjedrnih sistemov so asimetrični večjedrni sistemi, kjer so na enem čipu procesorska jedra različnih vrst in zmogljivosti. Na takšnih sistemih se praviloma izvaja ločen primerek operacijskega sistema na vsakem posameznem jedru, kar vprašanje komunikacije med opravili dodatno zaplete. Potrebno je namreč ustvariti ustrezen komunikacijski kanal med opravili, ki tečejo na različnih operacijskih sistemih. V tem diplomskem delu smo razvili mehanizem komunikacije med opravili na asimetričnem dvojedrnem krmilniku LPC4350 z uporabo skupnega pomnilnika. Pri tem smo preučili standardne mehanizme za usklajevanje in komunikacijo in uporabili njihove ključne koncepte pri zasnovi naše rešitve. Mehanizem komunikacije smo implementirali na operacijskem sistemu FreeRTOS in uporabili na praktičnem primeru preprostega temperaturnega krmilnika.

Ključne besede: asimetrični dvojedrni sistem, komunikacija med opravili, FreeRTOS, LPC4350, sistemi v stvarnem času, skupni pomnilnik

Abstract

The need for increasingly powerful computer systems on the one side and physical limitations on the other have brought about the development of multi-core processors which are becoming part of embedded systems and may often perform time-critical functions. It is particularly important that such systems involve a communication mechanism between various software tasks that need to be synchronised and able to exchange information in a reliable way and predictably in terms of time. Asymmetric multi-core processors belong to a special type of multi-core systems in which one chip combines several processor cores of different types and capacities. In case of such systems, separate operating systems are run on each individual core, which further complicates the communication between tasks. To this end, an adequate communication channel needs to be established between tasks running on different operating systems. For the purpose of the thesis we developed a communication mechanism between tasks on a dual-core controller LPC4350 by using shared memory. We also examined the standard mechanisms for synchronisation and communication and applied their key concepts in our solution. The communication mechanism was implemented on the FreeRTOS operating system and used on a practical example involving a simple temperature controller.

Keywords: asymmetric dual-core system, inter-task communication, real-time systems, LPC4350, FreeRTOS, shared memory

Poglavje 1

Uvod

1.1 Vseprisotno računanje in vgrajeni sistemi

Vseprisotno računanje je paradigma, ki računalniške sisteme različnih oblik in vrst vnaša v tako rekoč vse pore našega vsakdanjika. Morda bi lahko resnici na ljubo ta nekoliko pretenciozen uvod zapisali nekoliko drugače, kajti cilj vseprisotnega računanja je, da računalniške sisteme postavimo v tiste dele našega okolja, kjer nam lahko koristijo, pri tem pa nam ne smejo biti v napoto oziroma morajo biti čim bolj neopazni. Če se zazremo v razvoj računalniških sistemov tekom zadnjih dveh desetletij, bomo dejansko opazili številne spremembe, ki govorijo v prid tej naši trditvi. Če so se v devetdesetih letih prejšnjega stoletja v naših domovih šele dodobra uveljavili relativno veliki in okorni namizni računalniki, imamo sedaj velik razmah pametnih telefonov in tudi tabličnih računalnikov. Še bolj impresivno se zdi, ko v članku o vseprisotnem računanju iz leta 1993, ki ga je napisal M. Weiser iz raziskovalnega centra Xerox Palo Alto Research Center (PARC) [1], zasledimo prav takšno vizijo v takrat že delujočih prototipih tablic z zasloni na dotik.

To pa je šele začetek. Tehnološki razvoj je omogočil, da imamo v vsakdanji rabi čedalje več majhnih, energetsko učinkovitih in tudi relativno zmogljivih računalniških sistemov. Mnogi izmed njih služijo temu, da izboljšajo delovanje in uporabnost naprav, ki smo jih sicer že navajeni uporabljati. De-

nimo v hladilnikih, ki zaznavajo temperaturo in vlažnost zraka v posameznih predalih in skladno s tem uravnavajo svoje delovanje, s čimer podaljšajo obstojnost živil in hkrati zmanjšajo porabo energije. Še sodobnejši gredo korak dlje in zaznavajo hrano, ki je v njih shranjena. Tako nam lahko pomagajo pri pripravi nakupovalnega seznama ali pa nam celo predlagajo recept za pripravo obroka.

Nekateri drugi sistemi pomembneje vplivajo na našo varnost ali celo ohranjanje življenjskih funkcij. Naprednejši avtomobili so recimo sposobni prepoznavati prometne znake in opozarjati voznika na različne nevarnosti kot so npr. sprememba voznega pasu, premajhna varnostna razdalja ali nevarnost čelnega trčenja. Bolnikom z življenjsko nevarnimi motnjami srčnega ritma lahko danes vsadimo napravo majhnih dimenzij, ki s šibkimi električnimi stimulacijami uravnava ritem srca, ob nenadnem srčnem zastoju pa sproži močnejši elektrošok, s čimer poskuša spraviti srce nazaj v normalen ritem. Takšna naprava se imenuje *vsadni kardioverter-defibrilator* (kratica: *ICD*), saj združuje funkciji popravljanja ritma in defibrilacije.[1]

Vsi prej našteti primeri so sistemi, ki so specializirani za neko točno določeno funkcijo, čeprav so v grobem arhitekturnem smislu lahko dokaj podobni običajnim, splošno namenskim računalnikom. Tako eni kot drugi vsebujejo neko vrsto centralne procesne enote, glavni pomnilnik in eno ali več vhodno-izhodnih naprav. Po drugi strani pa je prav prilagoditev strojnih in programskih komponent teh specializiranih sistemov, ki jim pravimo tudi *vgrajeni sistemi*, ključna za doseganje želenih lastnosti, ki jih ti sistemi morajo izpolnjevati. Te lastnosti največkrat opredelimo z majhno dimenzijo, nizko porabo energije in primerno hitrostjo izvajanja funkcij.

Kako pomembne so lahko vse te lastnosti, lahko prikažemo na že omenjenem primeru vsadnega defibrilatorja. Prvo takšno napravo so človeku vsadili leta 1980 in že takrat je bila obetavna v smislu podaljševanja življenja bolnikov. Več kot očitne pa so bile njene pomanjkljivosti, ki so izhajale iz tehnoloških omejitev, značilnih za ta čas. Relativno visoka poraba energije

[1] Brez funkcije popravljanja ritma ji pravimo vsadni defibrilator.

(a) 209 cm^3 (b) 80 cm^3 (c) 36 cm^3

Slika 1.1: Primerjava velikosti različnih vsadnih defibrilatorjev

v kombinaciji s precej omejeno zmogljivostjo baterij je pomenila pogosto menjavo le-teh, kar je v praksi pomenilo ponovno vsaditev nove naprave. Zaradi omejenih zmožnosti procesiranja je bila naprava sposobna izvajati zgolj funkcijo defibrilacije, torej brez popravljanja srčnega ritma in tudi nekaterih dodatnih funkcij, ki so za današnjo generacijo naprav nekaj povsem običajnega. Med njimi naj omenimo npr. možnost programiranja in telemetrijo. Pomembni lastnosti, ki vplivata na zahtevnost posega vsaditve in kakovost življenja bolnika, sta masa in velikost naprave. Prostornina današnjih naprav znaša okoli 30 cm^3, za primerjavo pa je prva takšna imela prostornino 162 cm^3, njena masa pa je znašala 293 g. Za boljšo predstavo lahko na sliki 1.1 vidimo primerjavo naprav različnih velikosti. [2, 3]

1.2 Motivi za razvoj večjedrnih sistemov

Pregled zgodovine razvoja mikroprocesorjev kaže na precejšnjo odvisnost od napredka na področju tehnologije izdelave integriranih vezij. Pred več kot tremi desetletji je ekipa znanstvenikov iz IBM-ovega raziskovalnega centra T. J. Watson, pod vodstvom Roberta Dennarda, preučila možnosti manjšanja MOSFET tranzistorjev. Izdelali so smernice, s katerimi je možno zmanjševati

dimenzije tranzistorjev ob ohranjanju gostote moči in povečanju hitrosti delovanja. Te smernice vključujejo proporcionalno tanjšanje oksidnega polprevodnika (SiO_2) med vrati in kanalom, doziranje primesi v polprevodnikih tipa N in P in ustrezno zniževanje napajalne in pragovne napetosti tranzistorja. Vse to namreč ohranja jakost električnega polja v tranzistorjih, kar je zelo pomembno z vidika zanesljivosti delovanja, hkrati pa tudi občutno vpliva na porabo energije oziroma ohranjanje gostote moči. Če s $k < 1$ označimo faktor zmanjšanja tranzistorja naslednje generacije[2], potem se to na drugih lastnostih tranzistorja odraža na naslednji način:

Površina tranzistorja se zmanjša za faktor k^2. Ker računamo zmanjšanje površine, moramo upoštevati, da se tranzistor približno enako zmanjša v obeh dimenzijah.

Frekvenca se poveča za faktor $1/k$. Ker je tranzistor krajši, se proporcionalno zmanjša tudi čas potovanja signala in od tod višja frekvenca, ki je obratno sorazmerna s tem časom.

Napajalna in pragovna napetost se zmanjšata za faktor k. Da bi ohranili jakost električnega polja v tranzistorjih, moramo (po Dennardu) proporcionalno zmanjšati napajalno in pragovno napetost.

Kapacitivnost vrat tranzistorja se zmanjša za faktor k. Tudi ta posledica izhaja neposredno iz zmanjšanja dimenzije tranzistorja.

Če privzamemo $k = \frac{1}{\sqrt{2}}$, potem to pomeni zmanjšanje površine tranzistorja za faktor $k^2 = \frac{1}{2}$ na generacijo. To tudi ustreza Moorovemu zakonu, ki pravi, da naj bi se količina tranzistorjev na enoti površine ob vsaki naslednji generaciji podvojila glede na prejšnjo. Šele ob upoštevanju zgoraj naštetih lastnosti skaliranja MOSFET tranzistorjev pa Moorov zakon resnično pridobi na pomenu. Vse te lastnosti namreč pomembno vplivajo na moč, ki se

[2]Običajno gledamo faktor, za katerega se zmanjša dolžina vrat oz. razdalja med izvorom in ponorom.

v nekem vezju troši pri preklopih v tranzistorjih in jo izrazimo z naslednjo enačbo (1.1).

$$P = n * C_L * V^2 * f \tag{1.1}$$

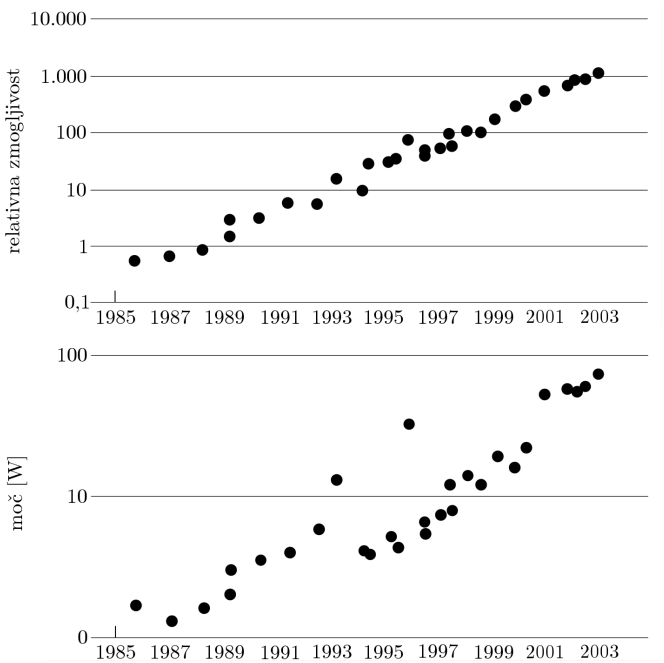
Ta pravi, da je moč P, ki jo neko integrirano vezje troši pri delovanju s frekvenco f, enaka produktu te frekvence s skupno kapacitivnostjo logičnih elementov in kvadratom napajalne napetosti V. Pri tem je C_L kapacitivnost posameznega logičnega elementa in n število vseh logičnih elementov v vezju.

Oglejmo si, kako je z močjo ob prihodu nove generacije integriranih vezij. Če ohranjamo konstantno velikost čipa in pridobljeni prostor zapolnimo z dodatnimi logičnimi elementi, bo število logičnih elementov sedaj enako $n * \frac{1}{k^2}$. Skalirajmo še vrednosti C_L, f in V iz prejšnje generacije skladno s prej izraženimi pravili za kapacitivnost, frekvenco in napetost. Sedaj lahko vidimo, da moč ostane enaka, saj je:

$$P = (n * \frac{1}{k^2}) * (C_L * k) * (V * k)^2 * (f * \frac{1}{k}) = n * C_L * V^2 * f \tag{1.2}$$

Kljub temu pa lahko vidimo precej strm trend naraščanja porabe energije mikroprocesorjev. Med leti 1985 in 2003 so zahteve po porabi energije skokovito narasle, kar lahko opazimo na sliki 1.2. [5] To je posledica tradicionalnega pristopa razvoja mikroprocesorjev, kjer so glavne pridobitve hitrosti dosežene s povečanjem frekvence delovanja vezja v kombinaciji z različnimi prijemi za vzporedno izvajanje ukazov. Treba je namreč upoštevati, da na frekvenco delovanja vezja ne vpliva zgolj hitrost preklopov v tranzistorjih, temveč tudi njegova zgradba. Nazoren primer so cevovodni procesorji, ki izvajanje ukaza razdelijo v več manjših stopenj. Ker se vsaka takšna posamezna stopnja lahko izvede v krajšem času, lahko frekvenco delovanja temu primerno povečamo.

Z nadaljnjim razvojem pa so se začele pojavljati tudi nove težave. Vedno daljši cevovodi in povečanje števila vzporednih cevovodov v superskalarnih procesorjih je pomenilo vedno kompleksnejšo logiko, tudi zaradi nujno po-



Slika 1.2: Trend naraščanja zmogljivosti in porabe energije mikroprocesorjev

trebnih mehanizmov za odpravljanje čedalje pogostejših cevovodnih nevarnosti. Kar zadeva porabo energije, pa je problem postal še posebej pereč od 90-nanometrskega procesa izdelave polprevodniških vezij dalje. Tu je namreč zaradi majhnosti tranzistorjev in zelo tankega dielektrika na vratih prišlo do pojava, ki je povzročil izgubo energije zaradi uhajanja električnega toka skozi vrata pri nizki pragovni napetosti. Temu toku pravimo uhajalni tok (angl. *leakage current*) in z manjšanjem tranzistorja vedno bolj narašča. Glede na to, da smo danes (2015) že v obdobju 14 nm procesa izdelave, si lahko predstavljamo, da problem resnično postaja zelo pereč in to kljub nekaterim tehnološkim inovacijam, ki so količino uhajalnega toka nekoliko omilile, a niti približno zmanjšale na raven pred 90 nm procesom.

Kopičenje vseh teh težav je spodbudilo oblikovanje novih pristopov za doseganje večjih zmogljivosti procesorjev. V zadnjem desetletju se je precej razširilo izkoriščanje vzporednosti na nivoju niti. Osnovna ideja tega pristopa je sledeča. Tok ukazov, ki jih procesor zaporedno jemlje iz glavnega

pomnilnika, razdelimo v dva ali več podtokov, ki jim pravimo *niti*. Procesor nato vsako posamezno nit izvaja vzporedno. Vendar pa je treba vedeti, da lahko pospešimo izvajanje le temu primerno napisane programske kode, saj moramo delitev na tokove ukazov na nek način opraviti že v času programiranja. To lahko storimo z različnimi programskimi konstrukti, kot so denimo procesi in programske niti. Če tega ne storimo, oziroma če je že sama narava problema takšna, da ne omogoča vzporednega reševanja podproblemov, potem z večnitnim izvajanjem ne bomo dosegli nikakršne pohitritve. Kako delež operacij v programu, ki se lahko izvajajo sočasno na c jedrih, vpliva na skupno pohitritev izvajanja, lahko ponazorimo z enačbo 1.3, ki izhaja iz Amdahlovega zakona.

$$N = \frac{1}{\frac{f}{c} + (1 - f)} \tag{1.3}$$

pri čemer je N faktor pohitritve programa, v katerem je delež operacij, ki se lahko izvajajo sočasno, enak f.

Vzporednost na nivoju niti lahko v strojni opremi dosežemo na več načinov. Danes je to najpogosteje z uporabo večjedrnih procesorjev, na katere lahko gledamo kot na več običajnih procesorjev na enem samem čipu. Posamično jedro v takšnem procesorju tako ustreza enemu ,,klasičnemu“ (enojedrnemu) procesorju. Ker je vsako jedro tako rekoč procesor s svojimi lastnimi viri, je zmožno izvajanja ločenega toka ukazov neodvisno od drugih jeder.

Z nekoliko naivnim razmišljanjem bi lahko predpostavili, da lahko povsem sočasno izvajamo toliko različnih oziroma med seboj neodvisnih programov, kolikor imamo jeder v procesorju in tako dosežemo faktor pohitritve, ki ustreza številu vseh jeder. Vendar ni tako. Ker imamo čedalje več sočasno izvajajočih se programov, bodo dostopi do ukazov in operandov v glavnem pomnilniku vedno bolj pogosti. Nekaj podobnega velja tudi za dostope do perifernih naprav. Vse te dostope je med seboj treba uskladiti, saj lahko do enega sistemskega vira, pa naj bo to glavni pomnilnik ali vhodno-izhodna naprava, praviloma dostopa le en procesor oziroma jedro hkrati. V vsaki takšni situaciji, ko naj bi prišlo do časovnega prekrivanja dostopov, je lahko

aktivno le eno jedro. Vsa ostala jedra, ki želijo dostopati do istega vira, bodo morala počakati, da aktivno jedro konča z dostopom.

Dodatna težava se lahko pojavi pri jedrih z vgrajenim predpomnilnikom. Če ima vsako jedro svoj lastni predpomnilnik, je treba zagotavljati koherenco med vsemi predpomnilniki in glavnim pomnilnikom. Rešitve tega problema obstajajo, a neizogibno vodijo v bolj zapleteno logiko in povečanje prometa na vodilu.

1.3 Večjedrni procesorji v vgrajenih sistemih

Ob vseh naštetih težavah je smiselno vprašanje, čemu tolikšna popularnost večjerdnih procesorjev. Odgovor leži v bolj učinkoviti porabi energije, saj zgradba večjedrnih procesorjev kljub naštetim zapletom omogoča večjo prepustnost pri isti porabi energije kot pri enojedrnem procesorju. Vendarle pa tudi z večjedrnimi procesorji ne moremo izničiti učinkov uhajalnega toka, temveč jih lahko kvečjemu omilimo. Industrija se je zato znašla v položaju, ki ga ni navajena, saj najverjetneje ne bo več mogla držati tempa večanja zmogljivosti procesorjev, ki si ga je tako rekoč sama vsilila. Proizvajalci zato iščejo različne nove priložnosti, s katerimi bi vsaj delno nadoknadili potencialno izgubo dohodka. To je tudi razlog, da se vedno več razvoja vrti okrog naprav v množični uporabi, ki v preteklosti niso imele nikakršnih zmožnosti procesiranja ali pa so bile le-te zelo omejene. Že podanim primerom avtomobilov, hladilnikov in srčnih defibrilatorjev lahko dodamo še mnoge druge, kot denimo prenosne telefone, ki so se razvili v male in relativno zelo zmogljive računalnike in v katerih so večjedrni procesorji danes prej pravilo kot izjema.

Ker gre pri večini teh naprav za vgrajene sisteme, lahko sedaj vidimo, kako je prihod večjedrnih procesorjev zaradi svoje večje učinkovitosti pri enaki porabi energije pomembno vplival na njihov razvoj. S stališča porabe energije so lahko še posebej zanimivi asimetrični večjedrni sistemi, kjer je procesor običajno zgrajen iz enega večjega in bolj zmogljivega jedra za splošne namene ter enega ali več manjših in med seboj enakih jeder, ki jih odlikuje

zelo nizka poraba energije in ki so običajno prirejena za določeno vrsto operacij. Veliko število majhnih jeder lahko izrabimo pri izvajanju vzporednih programov, medtem ko strogo zaporedni del pohitrimo tako, da ga izvajamo na bolj zmogljivem jedru kot bi ga sicer na simetričnem procesorju primerljive velikosti.

1.3.1 Uporaba v časovno kritičnih sistemih

Pri časovno kritičnih vgrajenih sistemih obstaja še en vidik, zaradi katerega se uporaba dvo- ali večjedrnega sistema zdi smotrna. Pri takšnih sistemih se namreč neredko zgodi, da imamo določen del spremljajočih funkcij, ki niso časovno kritične in morda zgolj nudijo neko dodatno udobje pri uporabi. Kopičenje takšnih funkcij pa neizogibno pomeni povečanje kompleksnosti programske rešitve, ki lahko posredno ogrozi pravilnost izvajanja kritičnih funkcij. Če bi programsko kodo razdelili na dva dela, pri čemer bi ločili kritične funkcije od preostalih, nato pa vsakega izmed njih popolnoma neodvisno poganjali na svojem jedru, bi lahko preprečili vplive izvajanja pomožnih funkcij na časovno kritične. Poleg tega bi pomembno zmanjšali kompleksnost kode, saj bi po novem vzdrževali dva povsem neodvisna programska sklopa. Kot dodatno prednost bi lahko izpostavili še lažji postopek verifikacije, ki je pri kritičnih sistemih običajno precej rigorozna, saj nepravilno delovanje lahko pomeni izgubo premoženja večje vrednosti ali celo človeškega življenja. Ker z izločitvijo raznih pomožnih funkcij zmanjšamo obseg kritičnega dela programske kode, je lahko postopek njene verifikacije temu primerno lažji in hitrejši.

Zgoraj opisana ideja se z vidika zanesljivosti zdi privlačna, vendar odpre dodatna vprašanja, na katera moramo odgovoriti. Prvo zadeva sočasni dostop do glavnega pomnilnika, saj si pri časovno kritični kodi običajno ne moremo privoščiti, da čaka na sprostitev vodila, medtem ko se na drugem jedru izvaja neka pomožna funkcija, ki to isto vodilo zaseda. To težavo lahko rešimo tako, da z ločenimi prenosnimi potmi vsakemu jedru zagotovimo dostop do svoje enote glavnega pomnilnika in tako odpravimo nevarnost blo-

kiranja izvajanja ukazov v enem jedru zaradi aktivnosti drugega jedra. Če imamo med procesorjem in glavnim pomnilnikom še predpomnilnik, pa moramo upoštevati, da zgrešitve v predpomnilniku slabo vplivajo na časovno predvidljivost izvajanja. To težavo najlažje rešimo tako, da predpomnilnika preprosto ne uporabljamo. [6, 11]

Problem komunikacije kot tema diplomskega dela

Naslednje vprašanje zadeva nivo programske opreme, kjer se lahko pojavi problem komunikacije in sinhronizacije med jedri. Če na primer neka kritična programska funkcija pripravi podatke, ki bi jih bilo treba zaradi nadaljnjega procesiranja dostaviti pomožni funkciji na drugem jedru, potrebujemo nek mehanizem, s katerim lahko te podatke učinkovito in v predvidljivem času prenesemo. To je tudi osrednje vprašanje, ki ga to diplomsko delo rešuje. Predstavili bomo izvedbo mehanizma komunikacije med opravili, ki se izvajajo na različnih jedrih asimetričnega dvojedrnega mikrokrmilnika LPC4350, pri čemer na vsakem jedru teče ločen primerek operacijskega sistema v stvarnem času. Za operacijski sistem smo izbrali FreeRTOS, ki je odprtokodni sistem, omogoča izvajanje opravil v stvarnem času, zaradi nizkih sistemskih zahtev pa je primeren za uporabo v številnih majhnih vgrajenih sistemih. Kljub svoji priljubljenosti [7] pa ne vsebuje nikakršnih podpornih funkcij za večjedrne sisteme. Z našim prispevkom bomo skušali to vrzel nekoliko zapolniti.

Poglavje Tehnologije in principi v prvem delu opisuje koncept stvarnega časa, ki zavzema osrednje mesto v sistemu FreeRTOS in je tudi sicer pogosto nepogrešljiv v vgrajenih sistemih. V drugem delu pa ponuja pregled najpogostejših mehanizmov medprocesne komunikacije, ki bodo služili kot navdih za našo rešitev.

Poglavje Uporabljene strojne in programske platforme opisuje strojno in programsko platformo, na kateri bomo razvili našo rešitev. Pri tem se v podrobnosti opisov spušča toliko, kolikor je potrebno za razumevanje delovanja in izvedbo rešitve.

Preostanek dela podrobno opisuje izvedbo rešitve na izbrani platformi, ki jo na koncu demonstrira na praktičnem primeru temperaturnega krmilnika.

Poglavje 2

Tehnologije in principi

2.1 Sistemi v stvarnem času

V Uvodu smo opozorili na obstoj sistemov, ki morajo opravljati časovno kritične naloge. Primerov takšnih sistemov je veliko in jih lahko najdemo med že omenjenimi, kot na primer sistemi za pomoč pri vožnji v avtomobilih. Nekako intuitivno bi lahko razbrali, da morajo tovrstni sistemi opravljati svoje storitve pravočasno, sicer so lahko rezultati njihovega delovanja povsem neuporabni, ali pa celo ogrozijo varnost. Če na primer avtomobil s funkcijo prepoznavanja prometnih znakov napačno ali prepozno prepozna prometni znak, bodo posledice napake običajno zanemarljive, saj gre zgolj za pomoč vozniku v obliki dodatnega opozarjanja na morebitno prekoračitev omejitve hitrosti ali katerega drugega prekrška. Če pa se bodo prototipi samovoznih avtomobilov izkazali kot dovolj dobri za vstop na trg, bo vloga računalniških sistemov v njih bistveno bolj pomembna. Tam več ne bo vseeno, če bo sistem prepozno zaznal nek objekt ali oviro na cesti, saj bodo vse z vožnjo povezane aktivnosti zaupane računalniku in se ne bomo več mogli zanesti na prisotnost in prisebnost voznika.

Seveda pa lahko ob vsem tem dodamo še nekaj dodatnih, za današnji čas bolj stvarnih sistemov, ki ne obstajajo le kot prototipi, temveč so v široki uporabi. Spomnimo se še enkrat na vsadni kardioverter-defibrilator, ki mora

pravočasno zaznati nepravilno delovanje srca. Temu dodajmo še sisteme za vzlet in pristajanje v letalih in ugotovili bomo, da smo lastno varnost morda že večkrat v svojem življenju zaupali računalniku.

2.1.1 Osnovne definicije

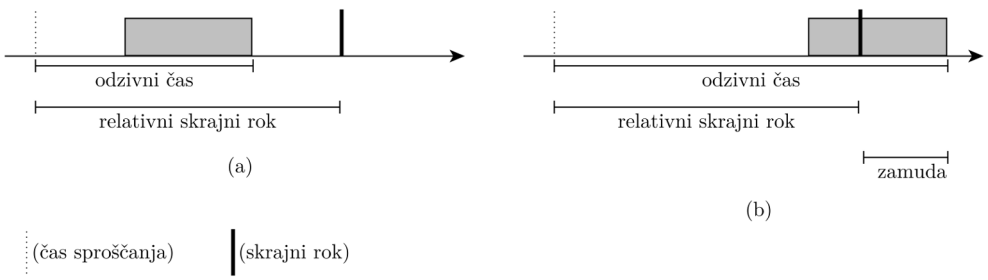
Zgornji opis sistemov v stvarnem času je seveda zelo ohlapen in nam pri njihovem razvoju ne pomaga prav veliko. Zato bomo vpeljali nekoliko bolj formalno definicijo teh sistemov. Še prej pa bomo razložili nekaj osnovnih pojmov, ki jih bomo pri tej definiciji potrebovali. [8]

Posli in opravila

Z besedo *posel* (angl. *job*) označujemo enoto dela, ki jo sistem razvrsti v izvajanje in tudi izvede. Množici povezanih poslov, ki skupaj opravljajo neko določeno funkcijo, pa pravimo *opravilo* (angl. *task*). Kot primer posla lahko navedemo vzorčenje izhodnega signala iz temperaturnega senzorja in preračunavanje vrednosti vzorca v temperaturo, izraženo s stopinjami Celzija. Nato imamo lahko periodično opravilo, ki tak posel sprošča v enakem, ustaljenem časovnih razmiku, ali drugače povedano ob izteku določene periode. Dodajmo še, da je vsak posel vezan na izvajanje na nekem sistemskem viru, ki mu običajno pravimo kar procesor.

Časovni parametri

Sedaj lahko vpeljemo nabor parametrov, s katerimi izrazimo časovne omejitve nad posameznim poslom. *Čas sproščanja* (angl. *release time*) je čas, ko posel postane na razpolago za izvajanje. *Skrajni rok* (angl. *deadline*) je čas, ko zahtevamo, da se izvajanje posla dokonča. Času, ki preteče od časa sproščanja do trenutka, ko se izvajanje posla dokonča, imenujemo *odzivni čas* (angl. *response time*). Maksimalen odzivni čas, ki je za nas še dopusten, imenujemo *relativni skrajni rok* (angl. *relative deadline*). Grafično ponazoritev teh dveh časov lahko vidimo na sliki 2.1, ki prikazuje dva primera izvajanja



Slika 2.1: Parametri časovnih omejitev

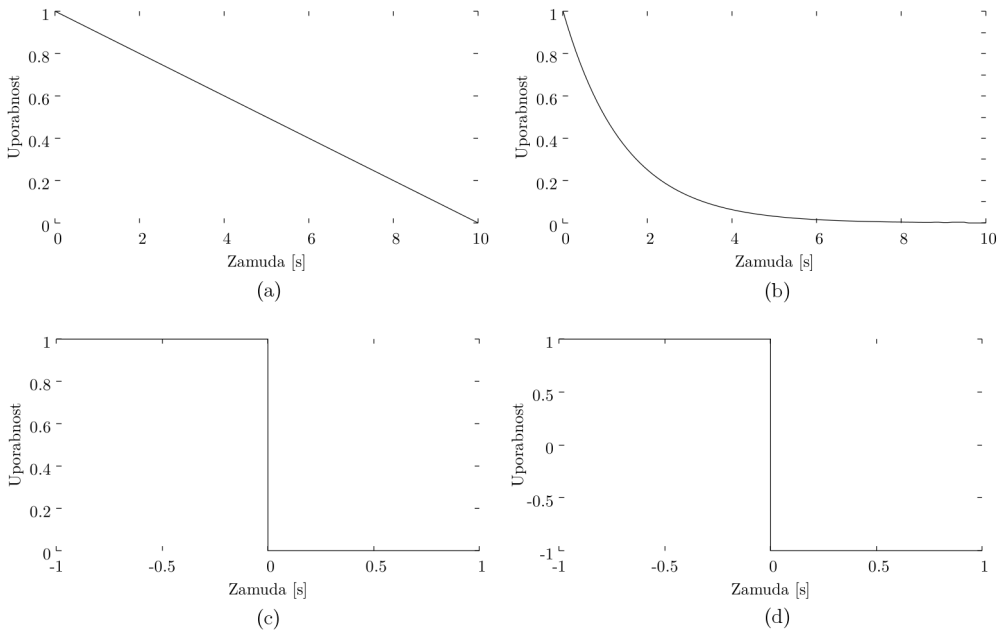
posla: enega, ko se opravilo prične izvajati dovolj zgodaj in ujame skrajni rok (a), in drugega, ko se izvede prepozno in nastane zamuda (b).

2.1.2 Časovne omejitve in sistemi v stvarnem času

Časovna omejitev je *stroga*, če njeno neizpolnitev ali prepozno izpolnitev smatramo kot usodno napako. Z drugimi besedami, četudi je rezultat, ki ga posel proizvede, točen, lahko njegovo prepozno dospetje povzroči dogodek, katerega posledice so lahko katastrofalne. Poslu s strogo časovno omejitvijo pravimo tudi posel v strogem stvarnem času.

Časovna omejitev je *mehka*, če njeno prepozno izpolnitev smatramo kot nezaželeno. Kot posledico neizpolnjene mehke časovne omejitve običajno razumemo poslabšanje učinkovitosti ali kakovosti storitve, ki lahko negativno vpliva na ugodje uporabniške izkušnje, nikakor pa ne more povzročiti materialne škode ali ogroziti varnosti ljudi. Poslom s tako podano časovno omejitvijo pravimo posel v mehkem stvarnem času.

Kot zanimivost omenimo, da časovno omejitev lahko opišemo tudi s funkcijo uporabnosti rezultatov posla (angl. *usefulness function*) v odvisnosti od zamude, pri čemer je zamuda (angl. *tardiness*) razlika v času med dokončanjem posla in njegovim skrajnim rokom. Nek posel tako zamuja, če je njegova zamuda večja od 0.



Slika 2.2: Primeri različnih funkcij uporabnosti

Na sliki 2.2 lahko vidimo grafe štirih primerov funkcij uporabnosti. Pri funkcijah (a) in (b) začne uporabnost padati ob pozitivni zamudi in doseže vrednost 0 ob zamudi 10 sekund. Tako definirano funkcijo, kjer uporabnost ne pade takoj ob pozitivni zamudi, običajno podamo pri poslih v mehkem stvarnem času. Drugače je pri funkcijah (c) in (d), kjer uporabnost pade takoj ob pozitivni zamudi. Takšne funkcije uporabnosti običajno zasledimo pri poslih v strogem stvarnem času. Pri funkciji (d) lahko vidimo, da je uporabnost ob pozitivni zamudi celo negativna (-1). Če takšen posel zamuja, potem je bolje, če ga sploh ne izvedemo, kajti negativna uporabnost rezultatov z drugimi besedami pomeni, da bo izveden posel povzročil škodo.

Funkcije uporabnosti služijo kot dober kvalitativni prikaz performančnih ciljev sistema v stvarnem času, saj lahko jasno vidimo, kako zamuda pri izvajanju posla vpliva na uporabnost rezultatov. Omenjamo jih predvsem zato, da dodatno prikažemo ločnico med posli v mehkem in strogem stvarnem času. V praksi pa redkeje zasledimo, da z njimi določamo časovne omejitve,

saj nam lahko otežijo postopek preverjanja pravilnosti delovanja sistema. Izjema so funkcije stopničaste oblike kot denimo na sliki 2.2 (c) in (d), vendar takšne časovne omejitve brez težav izrazimo tudi s parametri, ki smo jih že spoznali.

Končno se lahko vrnemo k vprašanju definicije sistema v stvarnem času. To je sistem, v katerem imamo vsaj en posel, za katerega je podana časovna omejitev. Če ima tak posel izraženo strogo časovno omejitev, potem govorimo o sistemu v strogem stvarnem času, sicer pa gre za sistem v mehkem stvarnem času.

2.2 Mehanizmi medprocesne komunikacije

2.2.1 Skupni pomnilnik

Eden izmed najpreprostejših mehanizmov medprocesne komunikacije je skupni pomnilnik. To je del glavnega pomnilnika, ki ga rezerviramo za uporabo v dveh ali več opravilih, ki želijo med seboj komunicirati. Ideja skupnega pomnilnika je preprosta - iz vsakega opravila naj bi bilo možno dostopati do podatkov v njem, pri čemer je način dostopa vedno enak, ne glede na to, iz katerega opravila opravljamo dostop. Gledano s programskega vidika naj bi ti dostopi potekali enako kot do različnih programskih spremenljivk, ki jih ima program shranjene drugod po glavnem pomnilniku. Zanje torej ne potrebujemo posebnih sistemskih klicev, podatke pa lahko pišemo in beremo neposredno, brez vmesnega kopiranja v nek medpomnilnik. Zaradi vseh teh razlogov je skupni pomnilnik tudi najhitrejši izmed vseh preostalih mehanizmov, ki jih bomo srečali v tem razdelku.

2.2.2 Cev

Cev je mehanizem medprocesne komunikacije, ki za prenos podatkov uporablja medpomnilnik velikosti N enot (običajno je ta enota bajt). Nad medpomnilnikom sta definirani funkciji *send* (pošlji) in *receive* (prejmi). Funk-

cija *send* pošlje enoto podatkov tako, da jo doda v medpomnilnik, funkcija *receive* pa deluje tako, da odstrani enoto podatkov iz medpomnilnika.

Motiv za uporabo

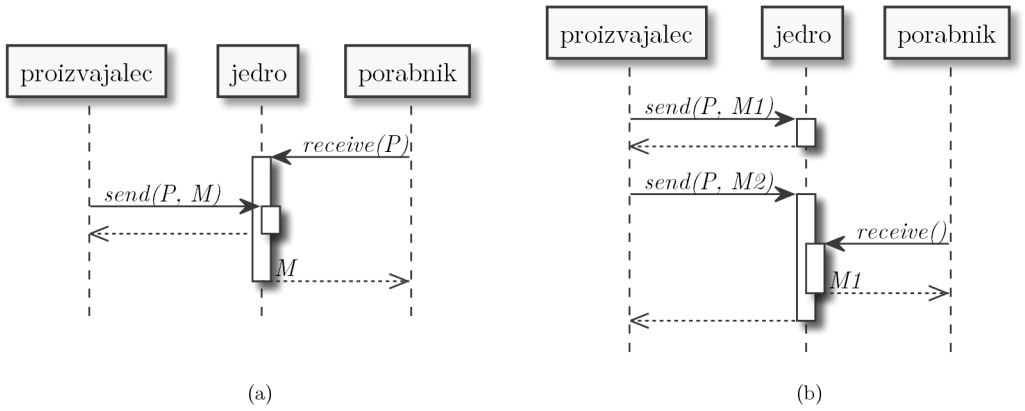
Vsakič, ko imamo dve opravili, ki med seboj komunicirata, rešujemo naslednji problem: kako naj izvajanje znotraj opravil uskladimo, tako da se operacije iz obeh opravil izvajajo v pravilnem vrstnem redu. Denimo, da vedno pošiljamo informacije iz istega opravila proti drugemu, ali drugače povedano, da je med opravili vzpostavljena polovična komunikacija (angl. *half duplex*). Imamo torej koncept proizvajalca in porabnika, pri čemer je proizvajalec opravilo, ki ves čas proizvaja informacije, porabnik pa jih porablja v smislu izvajanja različnih operacij, ki jih prispele informacije sprožijo. Sodelovanje med proizvajalcem in porabnikom moramo uskladiti tako, da proizvajalec ne preplavi porabnika z obilico podatkov, ki jih slednji ne utegne pravočasno procesirati. Prav tako moramo poskrbeti, da porabnik ne poskuša uporabiti nečesa, kar še ni bilo proizvedeno. Ti dve zahtevi lahko bolj formalno zapišemo kot dve precedenčni omejitvi oblike $a \preceq b$, s čimer povemo, da se dogodek a mora zgoditi pred dogodkom b.

Ker imamo opravka s proizvajalcem in porabnikom, bomo dogodek, ko proizvajalec pošlje i-ti podatek, označevali kot $send_i$. Podobno bomo za dogodek, ko porabnik prejme i-ti podatek, uporabili oznako $recieve_i$. Zapišimo sedaj precedenčni omejitvi za primer, ko imamo proizvajalca, ki pošilja podatke porabniku:

$$receive_i \preceq send_{i+1} \tag{2.1}$$

$$send_i \preceq receive_i \tag{2.2}$$

Prva omejitev določa, da moramo najprej prejeti i-ti podatek, preden lahko pošljemo naslednjega z indeksom $i+1$. Z drugo omejitvijo pa predpišemo, da i-tega podatka ne moremo prejeti, preden ga ne pošljemo. [9]



Slika 2.3: Blokiranje izvajanja funkcij nad cevjo

Prvo omejitev lahko nekoliko ,,razrahljamo“ s tem, ko dovolimo, da pošljemo še dodatnih $N-1$ podatkov po tem, ko smo poslali i-tega in ga porabnik še ni prejel. To zapišemo kot bolj splošno obliko omejitve 2.1:

$$receive_i \preceq send_{i+N} \tag{2.3}$$

Naslovljene težave

Cev je mehanizem medprocesne komunikacije, ki pravkar opisani koncept proizvajalca in porabnika na pregleden način prenese na nivo programskega vmesnika, saj imamo na voljo funkciji *send* in *receive*, ki ju lahko intuitivno uporabimo na mestih, kjer je treba prenašati podatke med procesi. Dodajmo še to, da je velikost medpomnilnika cevi enaka N v zapisu precedenčne omejitve 2.3.

Seveda je ob vsem tem pomembno, da je vrstni red prejemanja podatkov enak tistemu pri pošiljanju. To je zagotovljeno z ustrezno izvedbo medpomnilnika, ki je narejen tako, da zagotavlja FIFO oz. *prvi noter, prvi ven* (angl. *First-In-First-Out*) način dostopa. Tako bo prvi podatek, ki ga prejmemo iz cevi, enak tistemu, ki smo ga prvega poslali, oziroma splošneje je prejeti podatek tisti, ki je najdlje časa v medpomnilniku.

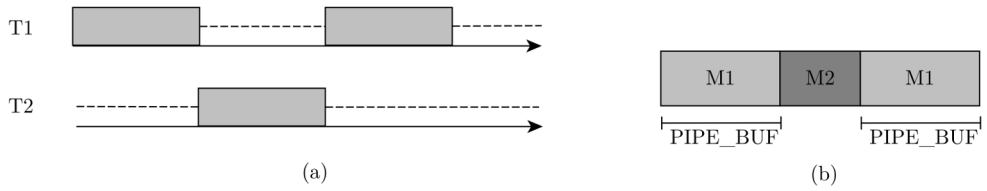
Da cev resnično realizira predpisani precedenčni omejitvi, imata funkciji *receive* in *send* pomembno lastnost, da blokirata izvajanje kličočega opravila, dokler ustrezni pogoj ni izpolnjen. Oglejmo si najprej scenarij, ki ustreza omejitvi 2.2 in ga dodatno ponazarjamo s sekvenčnim diagramom na sliki 2.3 a. Denimo, da je cev P prazna in da porabnik s funkcijo *receive* sproži prejem podatkov iz cevi P. Ker je medpomnilnik prazen, bo jedro operacijskega sistema blokiralo porabnika, vse dokler proizvajalec v cev P ne vpiše sporočila M.

Drugi primer (slika 2.3 b) prikazuje dogajanje, ki vsiljuje izpolnitev precedenčne omejitve 2.1. Tu proizvajalec v cev P najprej pošlje sporočilo *M1*, s ponovnim klicem funkcije *send* pa še *M2*. Jedro operacijskega sistema bo blokiralo dokončanje tega klica, saj je medpomnilnik poln in v njem ni več prostora za novo sporočilo. Izvajanje klica bo blokirano vse do trenutka, ko porabnik ne prebere starega sporočila *M1* in s tem sprosti prostor v medpomnilniku.

Prednost preproste uporabe cevi za komunikacijo med procesoma ima svojo ceno, saj v primerjavi z skupnim pomnilnikom izgubimo na hitrosti. Pri pošiljanju in prejemanju se namreč podatki kopirajo skozi medpomnilnik, poleg tega pa uporaba funkcij *send* in *receive* implicira uporabo sistemskih klicev za branje in pisanje podatkov. To ceno pa vseeno odtehta njihova uporabna vrednost, kajti zaradi preproste in intuitivne uporabe predstavljajo močno orodje za izmenjavo podatkov med opravili.

2.2.3 Sporočilna vrsta

Sporočilna vrsta je po principu delovanja zelo podobna cevi, večje razlike pa so na nivoju programskega vmesnika, ki uvaja nekaj več strukture nad vhodnimi in izhodnimi podatki. Cev tipično operira na bajtnem pretoku podatkov, oziroma pretoku z neko vnaprej predpisano granulacijo. Zato je bajt tudi teoretično minimalna zagotovljena količina podatkov, ki bo atomarno zapisana oziroma prebrana iz cevi. V praksi je ta količina sicer bistveno večja, se pa lahko precej razlikuje od izvedbe do izvedbe. Na sistemih Linux,



Slika 2.4: Primer razporejanja opravil pri pisanju v cev

ki implementirajo cevi po standardu POSIX.1-2001, je ta količina enaka 4096 bajtov, sam standard pa predpisuje dolžino vsaj 512 bajtov in jo označuje kot PIPE_BUF (oziroma njeno spodnjo mejo kot _POSIX_PIPE_BUF). [12] Pri sporočilni vrsti to količino definiramo sami, kajti vhodne in izhodne podatke sedaj predstavljajo atomarne enote, ki jih iz konteksta aplikacije definiramo sami in zajemajo vrednostno in časovno domeno enosmernega prenosa informacije. [13] Tem enotam pravimo sporočila, od tod tudi ime sporočilna vrsta za mehanizem, ki ga opisujemo v tem razdelku in ki ima vlogo osnovne storitve za prenos sporočil od pošiljatelja do enega ali celo več prejemnikov.

Motiv za uporabo

Kot bomo videli v nadaljevanju, so sporočilne vrste za razliko od cevi tipično zgrajene tako, da omogočajo sočasno uporabo iz več kot dveh opravil. Pri tem ni pomembno, ali imajo opravila vlogo proizvajalcev ali porabnikov.

Najprej si oglejmo, zakaj je sočasna uporaba iste cevi iz dveh ali več različnih opravil težavna. Če dve opravili istočasno pišeta v isto cev, se lahko zgodi, da razporejevalnik prekine izvajanje prvega opravila, medtem ko slednje vpisuje podatke v cev, nato pa začne z izvajanjem drugega opravila, ki bo nadaljevalo s pisanjem svojih podatkov v to isto cev. Posledica tega preklopa bo vidna na podatkih v cevi, saj bodo med seboj prepleteni. Do prepletanja sicer ne more priti na poljubnem mestu, saj imamo zagotovljen atomaren zapis PIPE_BUF bajtov in to je tudi dolžina, po kateri šele lahko pride do prepletanja.

Slika 2.4 a prikazuje primer razporejanja, ko opravilo T1 želi nepretrgoma vpisati sporočilo M1 dolžine dvakratnika PIPE_BUF bajtov, vendar pa razporejevalnik že po PIPE_BUF zapisanih bajtih preklopi izvajanje na opravilo T2, ki nato prične pisati svoje sporočilo (M2) v isto cev. Kako se takšno razporejanje izvajanja opravil kaže na vsebini medpomnilnika cevi, lahko vidimo na sliki 2.4 b, kjer se je v sporočilo M1 po prvih PIPE_BUF bajtih vrinilo še sporočilo M2. Težave torej nastopijo, ko imamo sporočila, ki so dolga več kot PIPE_BUF bajtov, pri čemer moramo vedeti, da je ta količina sistemsko določena in je ne moremo spreminjati. Poleg tega je lahko na vsakem sistemu drugačna in je zato toliko težje napisati platformno neodvisno aplikacijo.

Druga, sorodna težava je na strani porabnika. Zaradi preklopa izvajanja na drugo opravilo se lahko branje podatkov iz cevi predčasno prekine na poljubnem mestu. Torej nimamo nekega podobnega zagotovila kot pri pisanju, da bomo lahko nepretrgoma prebrali vsaj PIPE_BUF (ali morda neko drugo predpisano količino) bajtov. [12] Če razporejevalnik preklopi izvajanje na neko drugo opravilo, ki ravno v tem času prične z branjem iz cevi, se bo branje nadaljevalo na točki, kjer se je prejšnje opravilo zaustavilo. Tako bo to drugo opravilo prejelo del podatkov, ki so bili namenjeni prejšnjemu opravilu.

Dodajmo še to, da se tudi dolžine posameznih sporočil, ki jih pošilja nek proizvajalec, lahko razlikujejo in porabnik mora na nek način vedeti, koliko bajtov sme prebrati iz cevi, ne da bi prekoračil razmejitve med naslednjim sporočilom. Da bi rešili vse te težave, je treba na strani porabnika pa tudi proizvajalca implementirati dodatno logiko, ki bo upoštevala vse te pogoje. To je lahko precej zamudno in nerodno opravilo in bi zato bilo bolje, če bi bile te težave rešene že na nivoju mehanizma medprocesne komunikacije.

Naslovljene težave

Da bi rešili prej opisane težave z branjem in pisanjem, uvedemo atomarni princip pošiljanja in prejemanja sporočil. Tako izključimo možnost, da bo sporočilo le delno poslano ali prejeto, medtem ko sistemski razporejevalnik

preklopi izvajanje na drugo opravilo. Kljub temu pa moramo nekaj več pozornosti nameniti še eni dodatni težavi. Ko imamo več proizvajalcev, ki pišejo v isto sporočilno vrsto, se vsebine njihovih sporočil sedaj več ne morejo prepletati. Vendar pa je vrstni red, v katerem si njihova sporočila sledijo v sporočilni vrsti, odvisen od tega, kako se procesi proizvajalcev razporejajo v teku časa njihovega izvajanja. Ta vrstni red je lahko morda v nasprotju z želenim vrstnim redom.

Tipičen primer težave vrstnega reda dostave nastopi, ko imamo sporočila, ki se razlikujejo po pomembnosti. Ne glede na zasedenost vrste si želimo, da so bolj pomembna sporočila dostavljena pred manj pomembnimi. Drug primer je, ko imamo več porabnikov. Tu je vprašanje želenega vrstnega reda še posebej zapleteno, saj bo vrstni red pobiranja sporočil spet odvisen od tega, kako bo razporejeno izvajanje procesov porabnikov. V praksi pa so pogosti primeri, ko želimo, da so določena sporočila dostavljena točno določenemu porabniku.

Sporočilne vrste so običajno opremljene z ustreznimi mehanizmi, ki rešijo oba našteta problema. Najprej lahko vsakemu sporočilu dodelimo neko numerično prioriteto, na podlagi katere določimo vrstni red pobiranja sporočil iz vrste. Drug mehanizem je dodatna operacija, ki omogoča branje naslednjega sporočila iz vrste, ne da bi ga iz nje odstranili. Tej operaciji pravimo *peek* (slov. pokukaj). Tako lahko nek porabnik pregleda vsebino naslednjega sporočila in se na podlagi tega odloči, ali bo sporočilo pobral in ga obdelal, ali pa ga bo prepustil nekemu drugemu porabniku. Nekatere izvedbe sporočilnih vrst omogočajo tudi prirejanje enoličnega identifikatorja sporočilom. Na podlagi tega identifikatorja lahko sporočila pri pobiranju filtriramo in tako dosežemo, da porabnik celo brez operacije *peek* prejme zgolj tista sporočila, ki so v njegovi domeni.

Ker sporočilna vrsta v svoji izvedbi ni bistveno kompleksnejša od cevi, jo je razmeroma lahko vključiti v jedro praktično vsakega operacijskega sistema. To omogoča razmeroma tesno integracijo med sporočilno vrsto in razporejevalnikom, kar povečuje učinkovitost delovanja. Zato niti ni presenetljivo,

da jo kot osrednji mehanizem medprocesne komunikacije srečamo v večini operacijskih sistemov v stvarnem času, kot so FreeRTOS, VxWorks, QNX in drugi. [14] Prav zaradi pogoste uporabe v teh sistemih je sporočilna vrsta izbran mehanizem medprocesne komunikacije, na katerem bo zasnovana naša rešitev za komunikacijo v dvoprocesorskem sistemu.

Definicija

Za konec podajmo še nekoliko bolj formalno definicijo sporočilne vrste. [17] Sporočilna vrsta je vrsta, katere elementi so sporočila. Vrsta pa je kot podatkovna struktura dinamična množica z dvema kategorijama operacij: poizvedbe (npr. iskanje elementa po ključu) in spreminjajoče operacije (npr. dodajanje ali brisanje elementa). Gledano z vidika programske izvedbe so elementi dinamičnih množic tipično objekti, katerih atributi so dostopni preko referenc do teh objektov, nabor operacij pa je odvisen glede na to, kar dinamična množica predstavlja. Ko gre za vrsto, sta to vsaj naslednji dve operaciji:

Dodaj(Q, x). Doda element, na katerega kaže referenca **x** v vrsto **Q**.

Odstrani(Q). Odstrani vnaprej določeni element iz vrste **Q** in vrne referenco nanj. Ta element je določen kot prvi izmed obstoječih elementov v **Q**, ki smo ga dodali. Drugače povedano je to tisti element, ki je najdlje časa v vrsti **Q**. Tu gre torej za način dostopa FIFO, ki smo ga že izpostavili kot ključno lastnost cevi in sporočilnih vrst.

Ker je računalniški pomnilnik omejen vir, je tudi maksimalna velikost sporočilne vrste navzgor omejena, torej gre za končno dinamično množico. Navzgor je omejena tudi velikost posameznega elementa znotraj vrste.

2.2.4 Semafor

Semafor je programski konstrukt za sinhronizacijo, ki ga je med leti 1962 in 1963 izumil nizozemski znanstvenik Edsger W. Dijsktra. To je v osnovi celo

število, ki ga bomo označili kot S in nad katerim definiramo operaciji P in V[1]:

$P(S)$: Blokiraj izvajanje procesa, dokler vrednost semaforja S ni večja od nič, nato zmanjšaj vrednost semaforja za 1 in vrni.

$V(S)$: Povečaj vrednost semaforja S za 1.

S semaforji lahko uveljavljamo precedenčne omejitve. Če semafor S inicializiramo na vrednost n, potem z njim uveljavljamo omejitev:

$$V_i(S) \preceq P_{i+n}(S) \tag{2.4}$$

kjer je $V_i(S)$ dogodek, ki predstavlja i-to operacijo V nad semaforjem S (štejemo od trenutka, ko smo semafor inicializirali), $P_j(S)$ pa dokončanje j-te operacije P na tem istem semaforju. [10]

Problem omejenega vmesnika

Oglejmo si, kako lahko s pomočjo semaforjev uveljavljamo precedenčne omejitve 2.1, 2.2 in 2.3, ki smo jih našteli pri cevi. Če v omejitvi 2.4 privzamemo $n = N$ ter zamenjamo operaciji V z *receive* in P s *send*, dobimo:

$$receive_i \preceq send_{i+N}$$

kar je enako omejitvi 2.3. Očitno lahko semafor uporabimo kot števec prostih mest v medpomnilniku, saj je teh prav N, medtem ko funkcija za pošiljanje (*send*) ta števec zmanjša, saj opravlja vlogo operacije P. Podobno funkcija za sprejemanje (*receive*) opravlja vlogo operacije V in zato ta števec poveča. Z vidika porabe medpomnilnika je to povsem smiselno, saj ob pošiljanju dejansko zasedemo prostor v medpomnilniku, medtem ko ga ob prejemanju sprostimo. Obenem pa smo dosegli, da funkcija pošiljanja blokira vsakič, ko je medpomnilnik poln. Tako smo zadostili precedenčni omejitvi 2.3 (in tudi omejitvi 2.1 za posebni primer, ko je $N = 1$).

[1] Oznaki P in V izhajata iz nizozemskih besed *probeer* (poskusi) in *verhoog* (povečaj).

Nadaljujmo na podoben način in vzemimo še en dodaten semafor. Pri omejitvi 2.4 tokrat privzemimo $n = 0$, operacijo V zamenjajmo s *send* in operacijo P z *receive*. Dobimo:

$$send_i \preceq receive_i$$

kar je enako omejitvi 2.2. Ta semafor torej inicializiramo na vrednost 0 in predstavlja zasedenost medpomnilnika. Zato ga funkciji pošiljanja in prejemanja uporabljata na ravno obraten način. S tem, ko sporočilo pošljemo, zasedemo enoto medpomnilnika in zato funkcija pošiljanja na tem semaforju kliče operacijo V. Prejemanje sporočila po drugi strani sprosti enoto medpomnilnika, zato funkcija prejemanja sedaj kliče operacijo P. Na ta način smo torej izpolnili še precedenčno omejitev 2.2.

Problemu, ki smo ga pravkar rešili z uporabo dveh semaforjev, pravimo problem omejenega vmesnika (angl. *bounded-buffer problem*) in se v literaturi pogosto navaja kot zgled za prikaz uporabe sinhronizacijskih mehanizmov. [19]

2.2.5 Mutex

Pomembno vprašanje, na katerega pri semaforjih še nismo odgovorili, je atomarna izvedba operacije P. Kot vemo, mora ta operacija najprej preveriti, ali je vrednost semaforja večja kot 0. Če ta pogoj drži, mora potem vrednost semaforja zmanjšati za 1. Tako primerjava vrednosti kot odštevanje, ki ju potrebujemo za izvedbo operacije P, se morata izvesti atomarno. Opravila v tem času ne smemo prekiniti s preklopom na izvajanje nekega drugega opravila ali prekinitvene rutine, ki potencialno lahko izvede svoj klic operacije nad istim semaforjem, saj zaradi prepletanja ukazov za primerjavo in odštevanje lahko pride do nepravilnega vrstnega reda pri njihovem izvajanju in posledično do nepredvidljivega in nepravilnega delovanja. Delu programske kode, kjer potrebujemo takšno vrsto zagotovila, pravimo *kritično območje*.

Za reševanje problema sočasnega dostopa znotraj kritičnega območja lahko v resnici uporabimo kar binarni semafor (to je semafor, ki ga inicializiramo z $n = 1$). Pred vstopom v kritično območje kode oziroma programa kličemo operacijo P, po zaključku izvajanja znotraj kritičnega območja pa operacijo V. Takšnemu programskemu konstruktu pravimo *mutex* (okrajšava za *mutual exclusion*, izraz za *vzajemno izključitev* v angleščini) in kot lahko vidimo, gre za dokaj preprosto idejo. Vseeno pa imamo težavo že pri samem semaforju, saj tudi pri njegovi programski izvedbi nastopa kritično območje znotraj operacij P in V, ki jih moramo ustrezno ščititi. Obstaja kar nekaj strojnih in programskih rešitev te težave. Nekatere izmed njih si bomo ogledali v nadaljevanju tega razdelka.

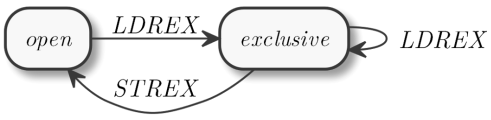
Izvedba z onemogočanjem prekinitev

Na enoprocesorskih sistemih lahko kritično območje ščitimo tako, da pred vstopom vanj onemogočimo prekinitve in jih nazaj omogočimo šele ob izstopu. Tako zagotovimo, da se nobena prekinitvena rutina ne bo izvedla v času izvajanja kritičnega območja. Ta rešitev je primerna tudi na večopravilnih operacijskih sistemih, kjer z izklopom prekinitev preprečimo tudi preklope med opravili. Slabost te rešitve je, da prispeva k večji zakasnitvi prekinitvenih rutin, obenem pa je neuporabna na večprocesorskih (in večjedrnih) sistemih.

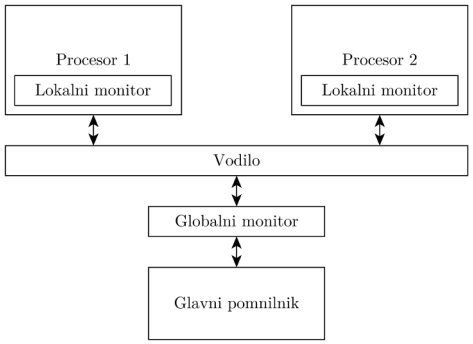
Izvedba s strojno podporo

Naslednja rešitev je možna le na nekaterih procesorjih in jo zato navajamo v obliki primera na strojni arhitekturi ARMv6. [18] Tu imamo na voljo končni avtomat, ki ga proizvajalec imenuje *ekskluzivni monitor*, in ki pozna dve stanji: *open* in *exclusive*. Poleg tega avtomata imamo na voljo še dva strojna ukaza: $LDREX$ (*load-exclusive*) in $STREX$ (*store-exclusive*).

Ukaz $LDREX$ zahteva dva operanda: ciljni register in pomnilniški naslov. Ta ukaz naloži vsebino pomnilniške besede iz podanega naslova v ciljni register, pri čemer ustrezno spremeni stanje ekskluzivnega monitorja, kot



Slika 2.5: Diagram prehajanja stanj lokalnega monitorja



Slika 2.6: Položaj ekskluzivnih monitorjev v dvoprocesorskem sistemu

bomo opisali v nadaljevanju.

Ukaz *STREX* zahteva tri operande: ciljni register, izvorni register in pomnilniški naslov. Če bo stanje ekskluzivnega monitorja ustrezno, bo ta ukaz shranil vsebino izvornega registra v pomnilniško besedo s podanim naslovom. Ne glede na ta pogoj pa bo v ciljni register vsakič vpisal vrednost, ki nam služi kot indikator, ali je bilo shranjevanje v pomnilnik opravljeno. Če je vrednost ciljnega registra enaka 0, potem vemo, da se je ukaz *STREX* uspešno izvršil. V nasprotnem primeru je vrednost tega registra enaka 1, kar pomeni, da je stanje ekskluzivnega monitorja preprečilo dokončanje ukaza *STREX*.

Procesor vsebuje primerek ekskluzivnega monitorja, ki mu pravimo *lokalni monitor*. Ko procesor izvrši ukaz *LDREX*, bo lokalni monitor prešel iz stanja *open* v stanje *exclusive*. Ukaz *STREX* za uspešno izvedbo zahteva, da je lokalni monitor v stanju *exclusive*. Ob dokončanju tega ukaza bo lokalni monitor prešel nazaj v stanje *open* (slika 2.5).

V primeru večprocesorskega sistema ima vsak procesor (ali jedro) svoj primerek lokalnega monitorja. Poleg teh imamo še en skupni ekskluzivni monitor, ki spreminja svoje stanje glede na to, ali kateri izmed procesorjev z ukazoma $LDREX$ oziroma $STREX$ dostopa do določenega pomnilniškega segmenta, ki je označen za skupno rabo med procesorji. Temu skupnemu ekskluzivnemu monitorju pravimo *globalni monitor* in njegov položaj v računalniškem sistemu lahko vidimo na sliki 2.6. V takšnem sistemu ukaz $LDREX$ spremeni stanje vseh monitorjev, ki so na poti do naslovljene pomnilniške besede. Podobno morajo biti pri ukazu $STREX$ vsi takšni monitorji v ustreznem stanju za njegovo izvršitev.

Globalni monitor se od lokalnega razlikuje po tem, da je za vsak procesor P_i sposoben hraniti po en par (P_i, A_i), kjer je A_i poljuben pomnilniški naslov, ki nastopa v ukazu $LDREX$ oziroma $STREX$. Globalni monitor označuje vsako takšen par tako, da ji priredi stanje *open* oziroma *exclusive*.

Če nek procesor P_x izvrši ukaz $LDREX$ z operandom na naslovu A_{x1}, bo globalni monitor zabeležil stanje *exclusive* za par (P_x, A_{x1}). Če procesor P_x sedaj znova izvrši ukaz $LDREX$, a tokrat z naslovom A_{x2}, bo globalni monitor počistil prejšnjo oznako tako, da bo za (P_x, A_{x1}) znova veljalo stanje *open*, za (P_x, A_{x2}) pa stanje *exclusive*. Če sedaj nek drug procesor P_y izvrši ukaz $LDREX$ na poljubnem naslovu A_y, bo globalni monitor označil tudi ta par (P_y, A_y), pri čemer bo še vedno ohranil oznako za (P_x, A_{x2}).

Oglejmo si še dogajanje v globalnem monitorju pri dveh možnih scenarijih ob izvajanju ukaza $STREX$ na procesorju P_x in z naslovom A_x:

- Če za (P_x, A_x) v globalnem monitorju velja stanje *exclusive*, potem se bo ukaz uspešno izvršil in globalni monitor bo počistil oznako za (P_x, A_x) in vse druge označene pare, v katerih nastopa naslov A_x, torej (P_i, A_x), $i \neq x$.

- V nasprotnem primeru, ko v globalnem monitorju ni oznake za (P_x, A_x), izvršitev ukaza ne bo uspešna.

Za uspešno izvedbo ekskluzivnega monitorja potrebujemo nek vstopni in

izstopni protokol, s pomočjo katerih zagotovimo zaščito kritičnega območja. Ta dva protokola moramo izvesti vsakič ob vstopu oziroma izstopu iz kritičnega območja. **ArmMonitorLock** in **ArmMonitorUnlock** sta primera takšnih protokolov, ki ju navaja proizvajalec in ki uporabljata prej opisane strojne konstrukte (ekskluzivne monitorje ter ukaza $LDREX$ in $STREX$). Pri tem je M celoštevilčna spremenljivka, ki jo imamo v glavnem pomnilniku in služi za hranjenje trenutnega stanja ekskluzivnega monitorja. To stanje je lahko bodisi *odklenjen* ali *zaklenjen*. Za vsako izmed teh stanj bomo uporabili neko svojo numerično vrednost. Točna izbira vrednosti ni pomembna, čeprav se običajno uporabljata števili 0 za odklenjeno in 1 za zaklenjeno stanje. Mi ju bomo označevali kar s simboli *Unlocked* za stanje *odklenjen* in *Locked* za stanje *zaklenjen*.

Algoritem ArmMonitorLock (vstopni protokol)

```
1: procedure ArmMonitorLock(M)
2: loop:
3:     r ← LDREX(M)
4:     if r = Unlocked then
5:         r ← STREX(Locked, M)
6:         if r = 1 then
7:             goto loop
8:     else
9:         goto loop
```

Vstopni protokol najprej z ukazom $LDREX$ naloži vrednost spremenljivke M v nek delovni register r in preveri stanje ekskluzivnega monitorja (vrstici 3-4). Če je že zaklenjen, ponovimo ta postopek od začetka (vrstici 8-9). V nasprotnem primeru poskusi z ukazom $STREX$ shraniti vrednost *Locked* v spremenljivko M. Če pri tem ne uspe, ponovimo ves postopek od začetka (vrstici 5-7). Iz prej podanih opisov obnašanja lokalnih ali globalnih monitorjev lahko ugotovimo, da bo do tega scenarija prišlo v naslednjih dveh primerih:

- drugo opravilo je na istem procesorju uporabilo ukaz $STREX$,
- drugo opravilo na različnem procesorju je uporabilo ukaz $STREX$ nad spremenljivko M.

Prvi primer očitno lahko povzroči nepotrebne iteracije znotraj protokola, saj je zaradi načina delovanja lokalnega monitorja popolnoma vseeno, ali je drugo opravilo uporabilo $STREX$ na isti ali različni spremenljivki. V drugem primeru pa je drugo opravilo očitno uspešno shranilo novo vrednost v M in s tem vstopilo v kritično območje, ki ga M ščiti.

Izstopni protokol je preprost, saj je vse, kar moramo narediti, le vpis vrednosti $Unlocked$ v M. Pri tem ne uporabljamo ukaza $STREX$, temveč običajen procesorski ukaz za pisanje v pomnilnik.

Algoritem ArmMonitorUnlock (izstopni protokol)

```
1: procedure ArmMonitorUnlock(M)
2:     M ← Unlocked
```

Petersonov algoritem

Kritično območje na dvoprocesorskem sistemu lahko ščitimo tudi s čisto programsko rešitvijo, brez posebnih konstruktov v strojni opremi. Opisali bomo rešitev, ki jo je leta 1981 predstavil Gary L. Peterson kot poenostavljeno različico Dekkerjevega algoritma. [20]

V skupnem pomnilniku hranimo dve spremenljivki. S prvo, ki jo bomo poimenovali *turn*, naznanimo opravilo, ki je na vrsti, da naslednje vstopi v kritično območje. Druga spremenljivka je polje dveh zastavic ($flags$), za vsako opravilo po eno. Posamično opravilo s svojo zastavico naznani, da želi vstopiti v kritično območje. Ti dve spremenljivki združimo v programsko strukturo, ki predstavlja mutex M.

V vstopnem protokolu (**PetersonLock**) opravilo z indeksom i najprej postavi svojo zastavico in tako naznani svojo namero o vstopu v kritično območje (vrstica 2). Nato nasprotnemu opravilu preda vrstni red tako, da

turn nastavi na indeks tega opravila (j, vrstica 3). V nadaljevanju opravilo i čaka, dokler nasprotno opravilo ne izstopi iz kritičnega območja ali dokler samo ni na vrsti (vrstice 4-6).

Algoritem PetersonLock (vstopni protokol)

```
1: procedure PetersonLock(M)
2:     M.flags[i] ← true
3:     M.turn ← j
4: loop:
5:     if (M.flags[j] = true) and (M.turn = j) then
6:         goto loop
```

Izstopni protokol je preprost. Opravilo i naznani svoj izstop iz kritičnega območja tako, da umakne svojo zastavico (**PetersonUnlock**, vrstica 2).

Algoritem PetersonUnlock (izstopni protokol)

```
1: procedure PetersonUnlock(M)
2:     M.flags[i] ← false
```

2.2.6 Monitor

Za konec si oglejmo še en konstrukt za sinhronizacijo, ki nam pomaga rešiti naslednji problem. Denimo, da imamo opravilo, ki v nekem trenutku vstopi v kritično območje in nato v njem čaka, dokler določeni pogoj ni izpolnjen. V tem času drugo opravilo (ali celo več njih) čaka na vstop v to isto kritično območje. Tako pridemo v situacijo, ko nobeno izmed omenjenih opravil ni dejavno v kritičnem območju, temveč vsa zgolj čakajo. Problem je še večji, če je za izpolnitev pogoja potrebno, da se funkcija iz drugega opravila izvrši v tem istem kritičnem območju.

Monitorji nudijo mehanizem, s katerim lahko opravilo začasno izstopi iz kritičnega območja in preide v stanje čakanja, pri čemer se postavi v čakalno vrsto. Sedaj lahko drugo opravilo vstopi v kritično območje in tam opravi

svoje delo. Če se ob tem izpolni pogoj, na katerega v vrsti čaka eno ali več opravil, lahko s pomočjo monitorja pošlje signal čakajočemu opravilu in takrat bo prvo izmed opravil v čakalni vrsti znova vstopilo v kritično območje in tam nadaljevalo z izvajanjem.

Na monitor tako lahko gledamo kot par mutex-a in seznama z naslednjimi operacijami:

wait: Začasno izstopi iz kritičnega območja in se dodaj v seznam čakanja.

signal: Odstrani prvo opravilo iz seznama čakanja in mu pošlji obvestilo, da se je zgodil opazovani dogodek.

enter: Vstopi nazaj v kritično območje in nadaljuj z izvajanjem. Ta operacija sledi operaciji *signal*.

Poglavje 3

Uporabljene strojne in programske platforme

3.1 FreeRTOS

3.1.1 Sporočilna vrsta

V poglavju Tehnologije in principi smo predstavili najpogosteje uporabljene mehanizme medprocesne komunikacije v računalniških sistemih. Spomnimo, da se na operacijskih sistemih v stvarnem času, med katerimi je tudi sistem FreeRTOS, za medprocesno komunikacijo v obliki sporočil najpogosteje uporablja sporočilna vrsta. [14] V istem poglavju smo našteli značilnosti sporočilnih vrst, ki jih bomo sedaj dodatno preučili na praktičnem primeru sistema FreeRTOS.

Opis programskega vmesnika

Izsek kode 3.1 prikazuje ključne funkcije iz programskega vmesnika sporočilnih vrst na sistemu FreeRTOS. Nov primerek vrste ustvarimo s funkcijo `xQueueCreate()`. Preko parametra `uxItemSize` podamo dolžino posameznega sporočila, s parametrom `uxQueueLength` pa maksimalno število sporočil, ki jih vrsta lahko vsebuje. Ta parameter lahko primerjamo s številom N, ki smo

ga srečali pri ceveh oziroma bolj natančno v precedenčnem pogoju 2.3, s tem da je sedaj ,,podatek“ opredeljen bolj točno in pomeni sporočilo.

Ko je vrsta enkrat ustvarjena, dobimo kot rezultat uspešne izvedbe funkcije `xQueueCreate()` programsko ročico do te vrste. Podatkovni tip programske ročice se imenuje `xQueueHandle` in ni nič drugega kot referenca do notranje podatkovne strukture, s katero je opisana sporočilna vrsta. Programsko ročico uporabljamo pri vseh preostalih funkcijah nad sporočilno vrsto in jo podamo kot prvi argument (`xQueue`). V prikazanem izseku kode so to vse funkcije za delo s sporočili (`xQueueSendToBack()`, `xQueueSendToFront()`, `xQueueReceive()` in `xQueuePeek()`), nenazadnje pa še funkcija za brisanje sporočilne vrste, ki sprosti vse z vrsto zasedene sistemske vire (`vQueueDelete()`).

Izsek kode 3.1: Osnovne funkcije sporočilnih vrst

```
1  xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength, unsigned
       portBASE_TYPE_t uxItemSize );
2
3  void vQueueDelete( xQueueHandle xQueue );
4
5  portBASE_TYPE xQueueSendToBack( xQueueHandle xQueue, const void *
       pvItemToQueue, portTickType xTicksToWait );
6
7  portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue, const void *
       pvItemToQueue, portTickType xTicksToWait );
8
9  portBASE_TYPE xQueueReceive( xQueueHandle xQueue, void *pvBuffer,
       portTickType xTicksToWait );
10
11 portBASE_TYPE xQueuePeek( xQueueHandle xQueue, void *pvBuffer,
       portTickType xTicksToWait );
```

V nadaljevanju si bomo ogledali podrobnejši opis funkcij za delo s sporočili in njihovih parametrov.

`xQueueSendToBack` (pošlji na konec). Doda sporočilo na konec sporočilne vrste. Ta operacija ustreza že opisanemu dodajanju v vrsto po načinu

FIFO.

`xQueueSendToFront` (pošlji na začetek). Doda sporočilo na začetek sporočilne vrste. Sporočilo, ki ga nazadnje dodamo na začetek vrste, bo prvo odstranjeno pri sprejemu. Ta način je ravno nasproten načinu FIFO in se imenuje *zadnji noter, prvi ven* oz. LIFO (angl. *Last-In-First-Out*). Ta operacija je nekoliko specifična za FreeRTOS sporočilne vrste in je ne srečamo v standardnih izvedbah sporočilnih vrst kot denimo System V ali POSIX, čeprav imajo tudi slednje različne mehanizme, s katerimi lahko vplivamo na vrstni red jemanja sporočil.

`xQueueReceive` (sprejmi). Vzame (odstrani) zadnje sporočilo iz sporočilne vrste. Konkretno katero sporočilo bo odstranjeno je odvisno glede na vrstni red sporočil v vrsti, ki ga določimo z zaporedjem izvajanja zgoraj naštetih operacij dodajanja.

`xQueuePeek` (pokukaj). To je izvedba operacije *peek*, ki smo jo že predstavili v razdelku Sporočilna vrsta poglavja Tehnologije in principi. Ta operacija je podobna operaciji sprejemanja s to razliko, da vrne kopijo sporočila, ne da bi ga odstranila iz vrste.

Poleg naštetih funkcij obstajajo še nekatere druge, pomožne operacije, kot denimo poizvedba po številu prostih mest v vrsti (torej razlika med številom vseh mest in številom zasedenih mest v vrsti). Ker poznavanje teh operacij bistveno ne prispeva k razumevanju zgradbe in delovanja sporočilne vrste na sistemu FreeRTOS, bomo njihovo obravnavo preskočili.

Prenašanje sporočil med sporočilno vrsto in opravilom

Pri formalni definiciji sporočilnih vrst na strani 28 smo omenili, da do elementov dinamičnih množic (oziroma v našem primeru sporočil) pri večini programskih izvedb dostopamo preko referenc. Oglejmo si še enkrat podpise funkcij, ki operirajo s sporočili, in se prepričajmo, da je tako tudi pri sporočilnih vrstah na FreeRTOS-u.

Funkciji `xQueueSendToBack` in `xQueueSendToFront` vzameta sporočilo preko vhodnega parametra `pvItemToQueue`, ki se prenaša po referenci. Podobno je tudi pri funkcijah `xQueueReceive` in `xQueuePeek`, kjer imamo izhodni parameter `pvBuffer`. Da gre za vhodni oziroma izhodni parameter, lahko vidimo že po deklaraciji parametra. Vhodni parameter `pvItemToQueue` vsebuje programski konstrukt `const` levo od simbola za referenco (`*`), kar pomeni, da ima funkcija zgolj bralni dostop do vsebine na naslovu, ki je določen s parametrom `pvItemToQueue`. Tega konstrukta seveda ni pri funkcijah `xQueueReceive()` in `xQueuePeek()`, saj za potrebe kopiranja sporočila na ciljno lokacijo potrebujeta pisalni dostop do nje.

Kljub vsemu moramo poudariti, da se vsebina sporočil v vrsto in iz nje prenaša po vrednosti in ne po referenci, kot bi lahko sklepali iz podpisov funkcij. Pri dodajanju sporočila v vrsto se tako ustvari njegova kopija, in sicer se prekopira vsebina pomnilnika v dolžini maksimalne velikosti sporočila iz naslova kot ga določa parameter `pvItemToQueue`. Podobno pri odstranjevanju s parametrom `pvBuffer` podamo lokacijo, kamor naj se prekopira sporočilo iz vrste. Z vidika porabe časa in pomnilnika je ta rešitev videti nekoliko potratna, a obstaja vrsta razlogov, zakaj se jo uporablja [21]:

Preprosta uporaba. Ko neko opravilo doda dinamično dodeljeni element v vrsto, mu ni treba skrbeti za življenjsko dobo tega elementa, saj vrsta sama poskrbi, da se pred dodajanjem najprej ustvari njegova kopija. Na ta način lahko opravilo kadarkoli po opravljeni operaciji dodajanja sprosti pomnilnik, ki ga zaseda izvirni element. Podobno velja tudi za elemente na skladu.

Zaščita pomnilnika. Na sistemih z enoto za zaščito pomnilnika oz. MPU (angl. *Memory Protection Unit*) lahko naslovni prostor razdelimo na več delov z različnimi pravicami dostopa. Tako lahko na primer del pomnilnika zaščitimo pred dostopi iz programske kode, ki ne teče v privilegiranem načinu delovanja. Opravila, ki tečejo v različnih načinih delovanja, lahko kljub temu med seboj še vedno komunicirajo. Če

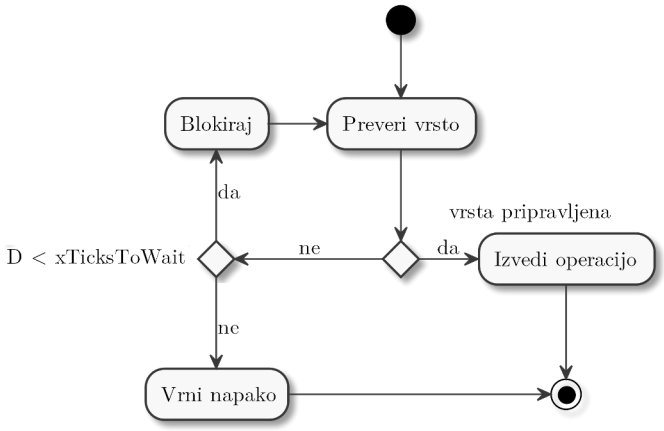
neko opravilo teče v uporabniškem (neprivilegiranem) načinu delovanja, lahko še vedno prejema sporočila tudi od privilegiranih opravil. Programska koda sporočilne vrste je namreč del jedrne kode, ki se izvaja v privilegiranem načinu in zato lahko dostopa do podatkov vsakega opravila. Ko neprivilegirano opravilo sproži operacijo sprejemanja sporočila, bo jedrni del prekopiral sporočilo v del pomnilnika, ki je dostopen opravilu samemu in ga slednje tudi poda s parametrom `pvBuffer`.

Prožnost. Podatke lahko v vrsto prenašamo tudi po referenci. To storimo tako, da elemente vrste definiramo kot reference na podatke v pomnilniku in nato te reference prenašamo po vrednosti. Da bi ta metoda delovala brez težav, pa moramo zagotoviti, da je del pomnilnika, v katerem se hranijo podatki, dostopen vsem opravilom, ki uporabljajo vrsto.

Časovne omejitve

Vse funkcije za delo s sporočili imajo še parameter `xTicksToWait`. S tem parametrom omejimo čas, ki ga lahko opravilo porabi s čakanjem na izpolnitev pogoja za uspešno izvedbo funkcije. Kako je ta pogoj izražen, je odvisno od funkcije same. Pri funkcijah za pošiljanje sporočila je to obstoj prostega mesta v vrsti za novo sporočilo. Pri funkcijah za sprejemanje je to prisotnost (vsaj enega) sporočila v vrsti. Če pogoj ni izpolnjen, bo izvajanje opravila blokirano do pojavitve enega izmed naslednjih dogodkov:

- Pretek `xTicksToWait` ciklov. Tej vrsti dogodka pravimo časovni dogodek.
- Dogodek, ki izvira iz nekega drugega opravila ali prekinitve. Tipično gre za enega izmed naslednjih dveh primerov: opravilo čaka na sprejem sporočila, medtem ko drugo opravilo pošlje novo sporočilo (1), ali pa opravilo pri pošiljanju čaka na prosto mesto v vrsti, medtem ko drugo opravilo sprejme sporočilo iz vrste (2). Tej vrsti dogodka pravimo sinhronizacijski dogodek [15] in ustreza dogajanju, ki smo ga opisali pri uporabi semaforjev za sinhronizacijo v razdelku 2.2.4 na strani 28.



Slika 3.1: Potek izvajanja blokirajoče funkcije sporočilne vrste

Po pojavitvi vsaj enega izmed zgoraj naštetih dogodkov bo opravilo ponovno pripravljeno na izvajanje. Ob pričetku ponovnega izvajanja se ponovno preveri pogoj za uspešno izvedbo operacije. Če ta pogoj še vedno ni izpolnjen, je postopek sledeč:

- če je od prvega blokiranja minilo `D` >= `xTicksToWait` ciklov, vrnemo napako;
- če je od prvega blokiranja minilo `D` < `xTicksToWait` ciklov, potem ponovno blokiramo opravilo za `xTicksToWait` − `D` ciklov.

Pravkar opisan potek izvajanja prikazuje tudi slika 3.1.

Parameter `xTicksToWait` lahko zavzame tudi dve posebni vrednosti. Če je `xTicksToWait` enak `0`, potem opravilo nikoli ne blokira, temveč funkcija v primeru neuspele operacije takoj vrne ustrezno kodo napake. Če je `xTicksToWait` enak `portMAX_DELAY`, potem lahko blokira nedoločen čas in tedaj ga v stanje pripravljenosti lahko spravi le sinhronizacijski dogodek.

Uporaba parametra `xTicksToWait` pomeni razširitev operacije *wait*, ki smo jo spoznali pri obravnavi semaforjev, saj omogoča časovno omejitev blokiranja izvajanja opravila. V primeru, da v danem času vrednost semaforja ne

preseže vrednosti nič, bo operacija neuspešna in to je tudi sporočilo opravilu, naj odneha od poskusa rezervacije vira.

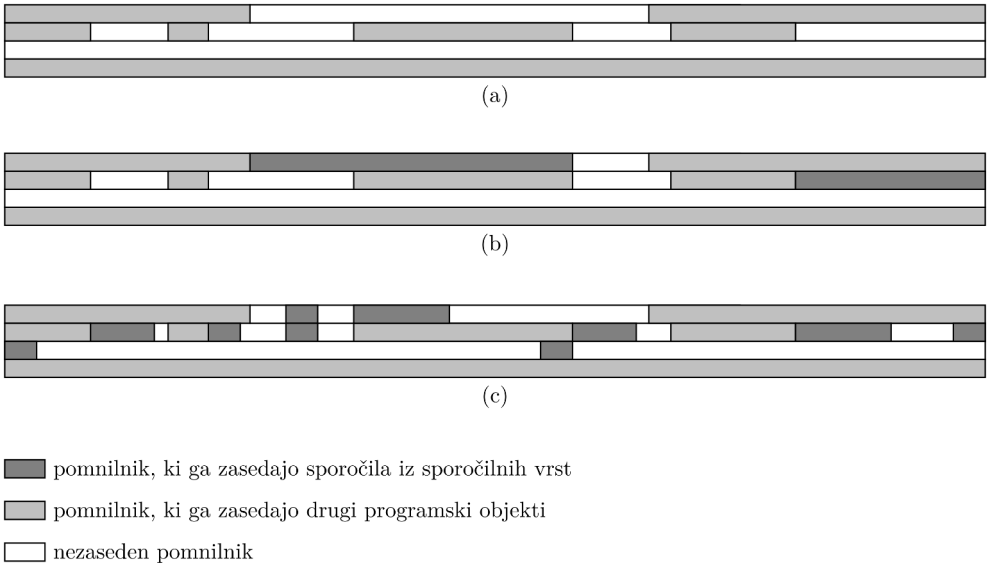
Delovanje semaforja je očitno testno vpeto v sporočilno vrsto. Razlago, zakaj je temu tako, lahko najdemo v razdelku 2.2.4 na strani 29. FreeRTOS to dejstvo izkorišča tako, da v istem programskem modulu združuje izvedbo sporočilne vrste in semaforja. Dodaten motiv za takšno odločitev je v kompaktnosti programske kode, kajti glede na to, da sporočilna vrsta uporablja semaforje za zagotavljanje precedenčnih omejitev, lahko na semafor gledamo kot na sporočilno vrsto z ničelno dolžino sporočila (parameter `uxItemSize`). Če je dolžina sporočila enaka 0, potem se sporočila skozi vrsto več ne prenašajo in nam ostane zgolj štetje prostih mest, seveda z mehanizmom sinhronizacije, ki smo ga že opisali. Tako funkcija za pošiljanje sporočila postane ekvivalent operacije *signal*, funkcija za sprejemanje pa operacije *wait*.

Notranja struktura

Pri pregledu programskega vmesnika sporočilne vrste smo spoznali, da vnaprej definiramo maksimalno število in velikost sporočil. Ta podatek je ob ustvarjanju vrste s pridom izkoriščen, saj se vnaprej rezervira dovolj pomnilnika za maksimalno število sporočil. Takšen pristop sicer lahko razumemo kot nekoliko potraten, saj bo prazna sporočilna vrsta zasedla enako količino pomnilnika kot polna. Obstaja pa več prednosti, ki so še posebej izrazite na sistemih v stvarnem času.

Najprej se izognemo naknadnim rezervacijam pomnilnika ob dodajanju novih sporočil v vrsto. Vsaka takšna rezervacija je namreč časovno potratna, saj zahteva iskanje nezasedenega bloka pomnilnika primerne velikosti. Poleg tega je ne moremo opraviti v konstantnem času in je časovno nepredvidljiva. Zaradi vsega tega bi operacija pošiljanja sporočila trajala dlje časa, hkrati pa bi povečali njeno časovno nepredvidljivost.

Dodatna težava, ki se ji z vnaprejšnjo rezervacijo pomnilnika v precejšnji meri izognemo, je fragmentacija pomnilnika. Na sliki 3.2 lahko vidimo ilustracijo tega problema. Slika 3.2 a predstavlja delno zaseden pomnilnik, v

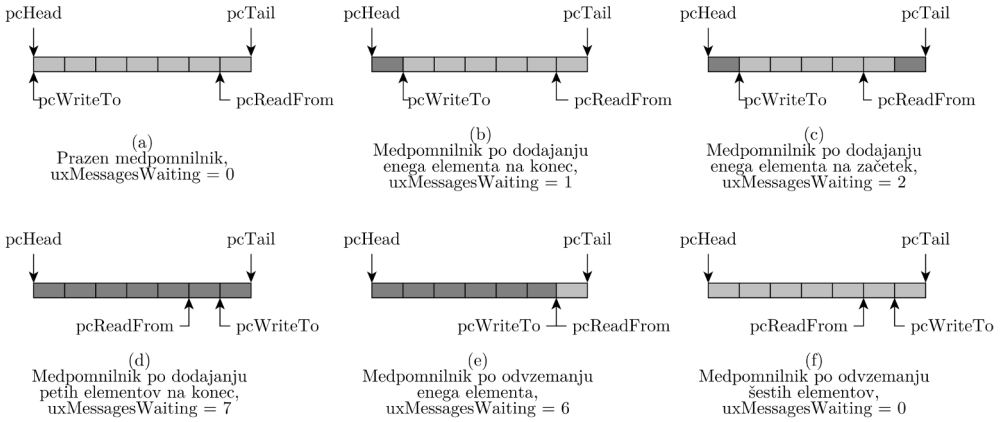


Slika 3.2: Prikaz možne zasedenosti in fragmentacije pomnilnika

katerem še ni nobene sporočilne vrste. Naknadno dodamo dve sporočilni vrsti različnih dolžin. Ob vnaprejšnji rezervaciji pomnilnika je pomnilnik lahko videti na primer tako, kot prikazuje slika 3.2 b. Takšen bo tudi ostal skozi vso življenjsko dobo sporočilnih vrst, ne glede na količino sporočil, ki jih v tem času prenesemo, ali pa zasedenost vrste v nekem trenutku. Če tega ne storimo, temveč rezerviramo pomnilnik za vsako sporočilo posebej, bomo imeli zaradi velikega števila rezervacij in sproščanj manjših pomnilniških blokov situacijo podobno tisti na sliki 3.2 c.

Sporočilna vrsta je na sistemu FreeRTOS realizirana kot krožni medpomnilnik. To je za način dostopa FIFO še posebej ugodno, saj se operaciji dodajanja in odstranjevanja izvajata v konstantnem času. Ker smo število sporočil in velikost posameznega sporočila navzgor omejili ob ustvarjanju vrste, je ves ta medpomnilnik sedaj vnaprej rezerviran, kar z vidika programske izvedbe pomeni dodatno preprostost in preglednost.

Sporočilna vrsta interno ohranja štiri reference: dve za označevanje začetka in konca medpomnilnika (skladno s poimenovanji v izvorni kodi FreeRTOS



Slika 3.3: Prikaz delovanja krožnega medpomnilnika

ju bomo označevali kot `pcHead` in `pcTail`), referenca na prvo zasedeno mesto in referenca na prvo prosto mesto (oznaki `pcReadFrom` in `pcWriteTo`). Poleg referenc vzdržuje še števec zasedenih mest (`uxMessagesWaiting`), ki ne more preseči maksimalnega števila elementov v vrsti. Namen uporabe števca je razločevanje med prazno in polno vrsto, kot je prikazano na sliki 3.3 (d) in (f).

Oglejmo si dogajanje v krožnem medpomnilniku ob pošiljanju sporočila na konec vrste. Sporočilo se najprej vpiše na prosto mesto, ki ga označuje referenca `pcWriteTo`. Ta referenca se nato ustrezno popravi, tako da kaže na naslednji element v medpomnilniku (slika 3.3 b). Ob vsaki spremembi reference je treba paziti, da le-ta ne prekorači območja, ki ga zaseda medpomnilnik. To je zagotovljeno tako, da se referenca `pcWriteTo` pred spremembo najprej primerja z mejno referenco `pcTail`. Če sta enaki, potem se referenca `pcWriteTo` izenači s `pcHead`. Na koncu se še števec zasedenih mest `uxMessagesWaiting` poveča za 1.

Pošiljanje sporočila na začetek vrste poteka na podoben način. Sporočilo se najprej vpiše na prosto mesto, ki ga označuje referenca `pcReadFrom`. Ta referenca se nato ustrezno popravi, tako da kaže na predhodni element v medpomnilniku (slika 3.3 c). Tudi tukaj je treba paziti, da ne pride do prekoračitve

medpomnilnika. Referenca `pcReadFrom` se zato pred pomikom nazaj primerja z mejno referenco `pcHead`. Če sta enaki, potem se referenca `pcReadFrom` izenači s `pcTail`. Na koncu se še števec zasedenih mest `uxMessagesWaiting` poveča za 1.

Sprejemanje sporočila iz vrste poteka tako, da se najprej popravi referenca `pcReadFrom`, tako da kaže na naslednji element v medpomnilniku (slika 3.3 e). Kot do sedaj je tudi tukaj treba paziti, da ne pride do prekoračitve medpomnilnika. Ker se `pcReadFrom` pomika naprej, se jo najprej primerja z mejno referenco `pcTail`. Če sta enaki, potem se jo izenači s `pcHead`. Na koncu se še števec zasedenih mest `uxMessagesWaiting` zmanjša za 1.

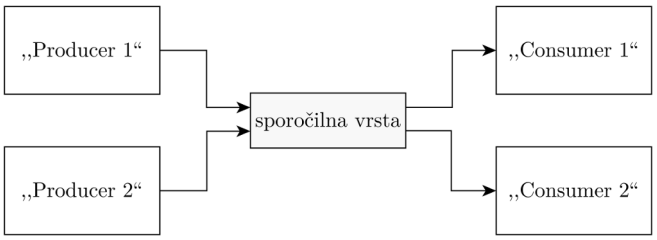
Primer uporabe

Pregled sporočilnih vrst na sistemu FreeRTOS bomo zaključili s konkretnim primerom za enoprocesorski sistem. Demonstrirali bomo prenos sporočil med dvema paroma proizvajalcev in porabnikov, ki med seboj komunicirajo preko ene same sporočilne vrste. Skupno imamo tako štiri opravila, med katerimi bomo proizvajalca poimenovali ,,Producer 1“ in ,,Producer 2“, porabnika pa ,,Consumer 1“ in ,,Consumer 2“ (slika 3.4).

Bistvo primera je prikazati način, s katerim na sistemu FreeRTOS ustvarimo sporočilno vrsto in jo kot programsko ročico prenesemo do različnih opravil. Obenem pa bomo demonstrirali možnost sočasne uporabe vrste iz več kot dveh opravil, ne glede na to, ali so opravila po svoji funkciji proizvajalci ali porabniki. Že v razdelku 2.2.3 smo povedali, da so sporočilne vrste običajno opremljene z mehanizmom, ki to omogoča. To velja tudi za sporočilne vrste na sistemu FreeRTOS. [15]

Izsek kode 7.1 (Dodatek, stran 103) predstavlja izvorno kodo našega primera. Funkcija `main()` ustvari sporočilno vrsto `xQueue` in vsa štiri opravila, nato pa še zažene razporejevalnik sistema FreeRTOS, s čimer se opravila pričnejo izvajati.

Opravilo ustvarimo s funkcijo `xTaskCreate()`, ki vzame šest parametrov v naslednjem vrstnem redu:



Slika 3.4: Primer prenosa sporočil med paroma proizvajalca in porabnika

Vstopna funkcija. Podati moramo referenco na funkcijo, ki naj bo klicana ob začetku izvajanja opravila. Programski vmesnik FreeRTOS zahteva, da ta funkcija nikdar ne vrne rezultata oziroma da vsebuje neskončno zanko.

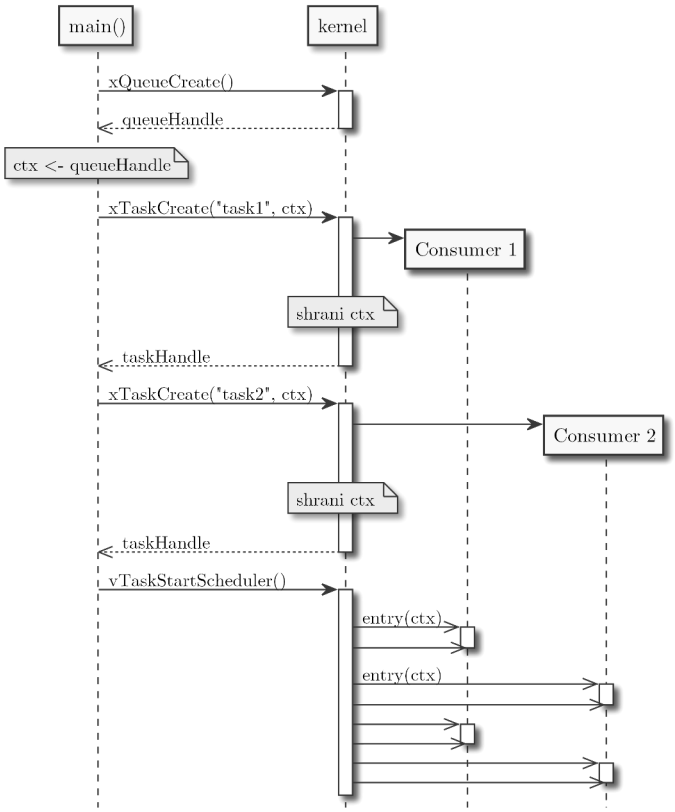
Ime opravila. Podamo ga kot niz znakov, katerega maksimalna dolžina je `configMAX_TASK_NAME_LEN`.

Globina sklada. Velikost sklada, ki bo dodeljen novemu opravilu. Podamo jo kot število besed.

Kontekst. S tem parametrom po referenci prenesemo poljubno podatkovno strukturo, ki bo podana naprej kot argument vstopne funkcije. Na ta način omogočimo, da ima opravilo neposreden dostop do takšne strukture.

Prioriteta. Numerična vrednost, s katero določimo prioriteto opravila. Ta ima lahko razpon od `0` do `configMAX_PRIORITIES - 1`, pri čemer večje število pomeni višjo prioriteto.

Programska ročica opravila. To je izhodni parameter, preko katerega dobimo programsko ročico opravila. S slednjo se lahko sklicujemo na ustvarjeno opravilo pri uporabi drugih funkcij za delo z opravili iz programskega vmesnika FreeRTOS. Uporaba parametra je opcijska in ga ignoriramo tako, da namesto reference na spremenljivko programske ročice podamo vrednost `NULL`.



Slika 3.5: Sekvenčni diagram ustvarjanja in razporejanja opravil

Ker je osnovna ideja programa pošiljanje sporočil od proizvajalcev do porabnikov, moramo dostaviti programsko ročico sporočilne vrste vsem sodelujočim opravilom. To storimo preko prej opisanega parametra za kontekst funkcije `xTaskCreate()`. Vrstice 107-110 izseka kode 7.1 prikazujejo, kako to storimo za oba porabnika. Tu kot parameter za kontekst uporabimo kar ročico `xQueue`, s čimer bo neposredno prenesena do vstopne funkcije vsakega izmed porabnikov.

Ker bo vsak proizvajalec pošiljal sporočila točno določenemu porabniku, mu moramo na nek način dostaviti še ime porabnika, ki mu bo naslovil sporočilo. Tudi to ime bomo prenesli preko kontekstnega parametra, skupaj z ročico do vrste. Točen način, kako to storimo, je prikazan z vrsticami 8-12 in 90-103. Najprej definiramo programsko strukturo `struct xProducerContext`, v

katero shranimo oba podatka, torej ime in ročico. Nato naredimo za vsakega proizvajalca svoj primerek te strukture in jo definiramo tako, da vsebuje ročico do vrste in ime porabnika (vrstice 90-91 in 99-100). Nazadnje to strukturo preko kontekstnega parametra prenesemo po referenci do vstopne funkcije proizvajalca (vrstice 93-94 in 102-103).

Kako točno se kontekst prenese do vstopne funkcije opravila, bomo lažje razumeli s pomočjo sekvenčnega diagrama na sliki 3.5. Ta diagram prikazuje prej opisano zaporedje ustvarjanja vrste in dveh porabnikov[1], pri čemer lahko sedaj bolje vidimo vlogo, ki jo ima pri tem operacijski sistem.

V diagramu vidimo, da se funkciji `xQueueCreate()` in `xTaskCreate()`, ki ju kličemo iz funkcije `main()`, izvajata v jedru operacijskega sistema (označeno kot ,,kernel“). Ko vrsto ustvarimo, pridobimo njeno programsko ročico in si jo shranimo s spremenljivko ,,ctx“. To spremenljivko nato uporabimo kot parameter za kontekst vsakič, ko ustvarimo novega porabnika. Znotraj funkcije `xTaskCreate()` si bo jedro za pravkar ustvarjeno opravilo shranilo kontekstni parameter za kasnejšo rabo.

Ko smo končali z ustvarjanjem opravil, zaženemo razporejevalnik s klicem funkcije `vTaskStartScheduler()`. Od tega trenutka dalje razporejevalnik v jedru prične z razvrščanjem in izvajanjem poslov opravil. Vsak posel je v diagramu prikazan kot klic do svojega opravila. Prvi posel vsakega opravila je sestavljen iz klica vstopne funkcije, katere parameter je kontekstna spremenljivka. V diagramu to ponazorimo kot sporočilo ,,entry(ctx)“. Tu sedaj vidimo, kako jedro uporabi predhodno shranjen kontekstni parameter. Nadaljnji posli so odvisni od narave samih opravil in za ta primer niso posebej pomembni.

Končno si oglejmo strukturo sporočila, ki se prenaša med proizvajalcem in porabnikom. V izseku kode jo najdemo v vrsticah 1-6 (`struct xTaskMessage`). Polja v strukturi so naslednja:

`cReceiver`: ime opravila, ki mu je sporočilo namenjeno,

[1] V diagram bi lahko dodali tudi proizvajalce, a s tem ne bi nič bolje pojasnili vloge operacijskega sistema in načina prenosa konteksta do opravil.

`cSender`: ime opravila, ki je pošiljatelj sporočila,

`lValue`: numerična vrednost, ki jo generira proizvajalec in je namenjena porabniku.

Preostane nam še razlaga vstopnih funkcij opravil. Vstopna funkcija za proizvajalce je `vProducer()` (vrstice 14-40). Ta v neskončni zanki dodaja v vrsto nova sporočila po načinu FIFO. V vsaki novi iteraciji priredi polju `lValue` vrednost, ki je enaka števcu iteracij. Ker se sporočilo vsakič pošlje iz istega opravila, se polje `cSender` nastavi še pred zanko (vrstica 24) in velja skozi ves čas izvajanja opravila. Ime opravila pošiljatelja je enako imenu opravila, ki se trenutno izvaja in ga dobimo s pomočjo funkcije `pcTaskGetTaskName()` programskega vmesnika FreeRTOS.

Vstopna funkcija porabnikov je `vConsumer()` (vrstice 42-74). Funkcija v neskončni zanki z operacijo *peek* najprej pogleda naslednje sporočilo iz vrste in primerja ime naslovnika z imenom lastnega opravila. Če se imeni ujemata, potem je sporočilo namenjeno trenutno izvajajočemu se porabniku in ga poberemo iz vrste, nato pa še izpišemo njegovo vsebino preko serijskega vmesnika.

Gledano z vidika uporabnosti je prikazani primer zelo preprost, a smo zato poskušali čim bolj nazorno demonstrirati uporabo ene sporočilne vrste iz več kot dveh opravil, pri čemer smo si podrobneje ogledali mehanizem prenosa programske ročice sporočilne vrste različnim opravilom. V naslednjem poglavju bomo spoznali, da moramo v heterogenem večjedrnem sistemu uporabiti nek drug mehanizem za dosego tega cilja. S prikazanim primerom se lahko tudi prepričamo, da je uporaba sporočilne vrste iz opravil preprosta, saj nam ni treba skrbeti za pobiranje sporočil po delih in njihovo medsebojno prepletanje - težavi, na kateri smo posebej opozorili v razdelku 2.2.3 (Sporočilna vrsta). Iz primera lahko vidimo še prednost prenašanja vsebine sporočil v vrsto po vrednosti (stran 42), kajti nikjer v kodi nam ni treba paziti na življenjsko dobo sporočil.

3.2 Mikrokrmilnik LPC4350

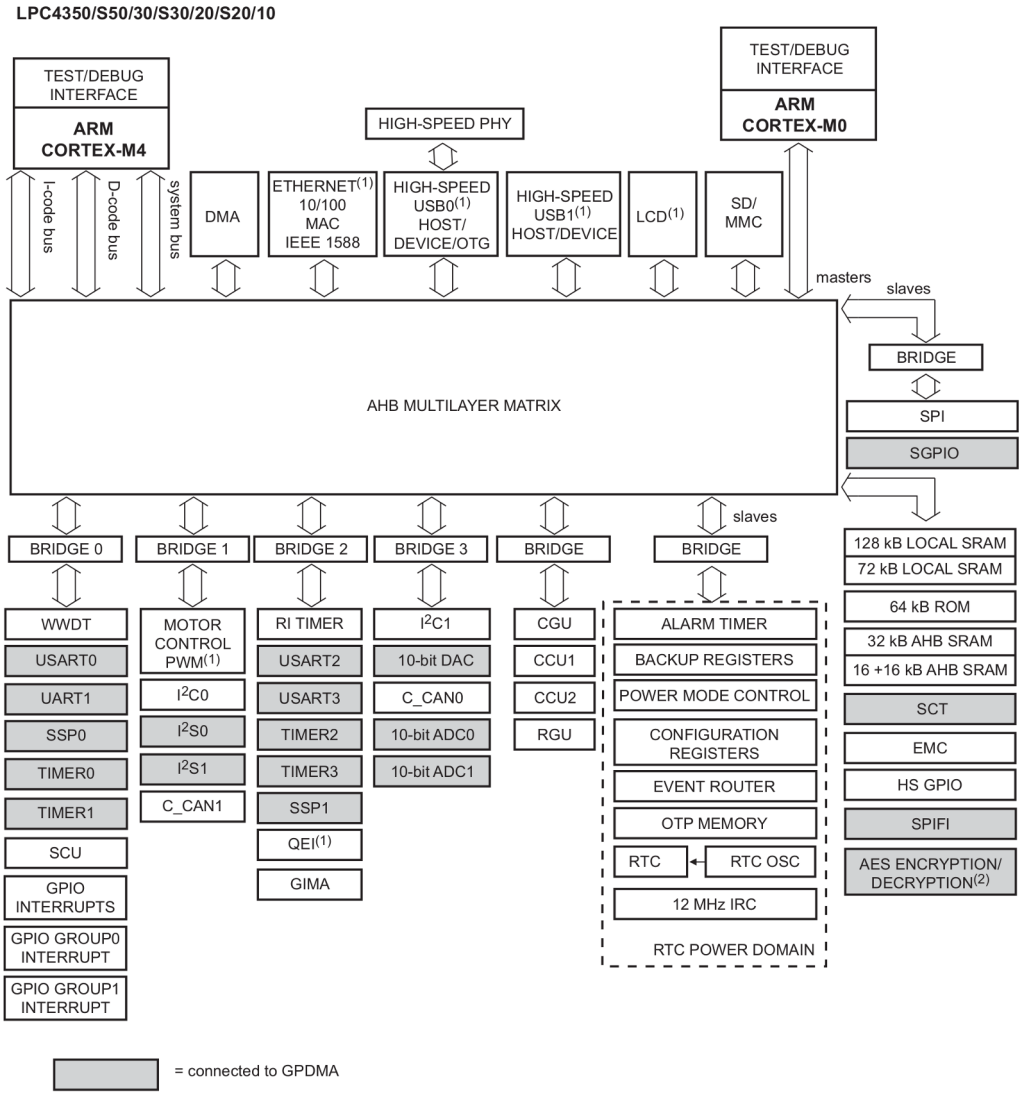
3.2.1 Opis arhitekture

LPC4350 je dvojedrni mikrokrmilnik, v prvi vrsti namenjen vgrajenim aplikacijam za digitalno procesiranje signalov. Glavno jedro je ARM Cortex-M4, pomožno pa ARM Cortex-M0. Oglejmo si nekaj poglavitnih značilnosti teh dveh jeder.

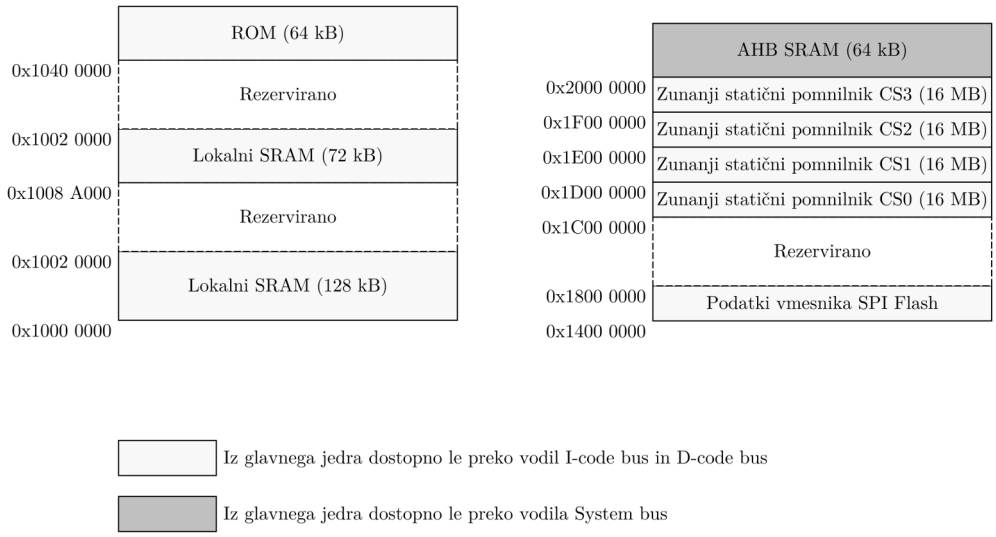
Glavno jedro deluje na frekvenci do 204 MHz, ima vgrajeno enoto za zaščito pomnilnika (MPU, angl. *Memory Protection Unit*), ugnezdeni vektorski prekinitveni krmilnik (NVIC, angl. *Nested Vectored Interrupt Controller*) in enoto za delo v plavajoči vejici.

Pomožno jedro služi kot koprocesor, saj je namenjeno temu, da na nek način razbremeni glavno jedro pri računsko intenzivnih operacijah, pa tudi pri delu z vhodno-izhodnimi napravami. Tudi to jedro deluje na frekvenci do 204 MHz in ima vgrajen ugnezdeni prekinitveni krmilnik, nima pa enote za delo v plavajoči vejici in enote za zaščito pomnilnika. Tudi nabor ukazov, ki je sicer navzgor združljiv s Cortex-M4, je manjši.

Glavno jedro ima harvardsko arhitekturo, pri kateri imamo ločen pomnilnik za ukaze in operande. [16] Zaradi ločenega pomnilnika seveda potrebujemo dvoje vodil, saj jedro v nasprotnem primeru ne more istočasno dostopati do obeh pomnilnikov. Diagrami iz tehnične specifikacije mikrokrmilnika na slikah nam razkrijejo nekaj več podrobnosti. Na sliki 3.6 je bločni diagram komponent mikrokrmilnika. Tu vidimo, da ima glavno jedro ne le dva, temveč tri vodila, ki so poimenovana kot *I-code bus*, *D-code bus* in *System bus*. Preko teh treh vodil lahko jedro istočasno dostopa do različnih podatkov, le-ti pa morajo biti ločeni glede na njihovo vrsto in segment v naslovnem prostoru, kjer se nahajajo. Konkretno to pomeni, da jedro uporablja vodilo *I-code bus* za dostop do ukazov, ki se nahajajo na naslovih 0x00000000 - 0x1FFFFFFF. Jedro za dostop do operandov v tem istem segmentu uporablja vodilo *D-code bus*, za dostop do vseh vrst podatkov (torej tako ukazov kot operandov) na naslovih od 0x20000000 dalje pa uporablja vodilo *System bus*. Vsa tri



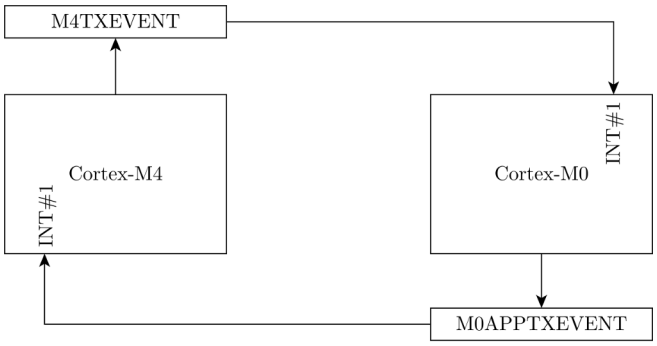
Slika 3.6: Bločni diagram mikrokrmilnika LPC4350



Slika 3.7: Del naslovnega prostora mikrokrmilnika LPC4350

vodila so priključena na osrednjo povezovalno komponento, to je večplastna matrika AHB (*AHB Multilayer Matrix*), ki služi medsebojnemu povezovanju več vodil AHB in s tem tudi vseh gospodarjev in sužnjev, ki so priključeni na ta vodila. Naprave, ki so gospodarji, so na sliki označene kot *masters*. To so vse naprave, prikazane na zgornji strani *AHB Multilayer Matrix* in med njimi sta seveda tudi obe jedri.

Mikrokrmilnik vsebuje 264 kB pomnilnika SRAM, na katerega lahko gledamo kot na več manjših pomnilnikov SRAM, ki so samostojno priključeni na vodilo. Na sliki 3.6 lahko vidimo, da je ta pomnilnik razdeljen na štiri enote, in sicer na 128 kB, 72 kB (obe nosita oznako *LOCAL SRAM*) ter dve po 32 kB (oznaka *AHB SRAM*). Ta delitev se dodatno vidi v prikazu naslovnega prostora na sliki 3.7, kjer je prva enota (128 kB SRAM) na naslovih 0x10000000 - 0x10020000, naslednjih 72 kB na 0x1008A000 - 0x10092000, AHB SRAM pa na 0x20000000 - 0x20010000. Prikaz naslovnega prostora nam nudi boljši vpogled v dosegljivost teh enot. Ker sta prvi dve enoti pomnilnika SRAM (128 kB in 72 kB SRAM) na naslovih manjših od 0x20000000, lahko glavno jedro do njih dostopa preko dveh različnih vo-



Slika 3.8: Povezava med jedri

dil, to sta *I-code bus* in *D-code bus*. To pomeni, da lahko ukaze in operande shranimo v poljubno izmed dveh enot pomnilnika SRAM. Da bi izkoristili pohitritev, ki jo omogoča arhitektura, moramo zato sami poskrbeti, da programsko kodo in podatke naložimo v različne enote. Če na primer naložimo tako programsko kodo kot podatke v prvo, 128 kB veliko enoto, potem jedro več ne bo moglo istočasno dostopati do ukazov in operandov.

Pri pomožnem jedru imamo princetonsko arhitekturo, saj imamo eno samo vodilo, s katerim lahko naenkrat dostopa samo do enega izmed naštetih pomnilniških enot SRAM. Preko tega vodila se torej prenašajo tako ukazi kot operandi. Kar zadeva povezljivost do drugih naprav, je pomožno jedro praktično enakovredno glavnemu jedru. Pomožnemu jedru so vidne tudi vse enote pomnilnika SRAM in tako predstavljajo skupni pomnilnik. Ta značilnost je pomembna za učinkovito komunikacijo med jedri, saj je skupni pomnilnik v mikrokrmilniku hiter in omogoča programsko izvedbo v prejšnjem poglavju naštetih mehanizmov medprocesne komunikacije. Kar je najpomembneje, pa lahko sedaj s programskimi mehanizmi medprocesne komunikacije povežemo procese, ki tečejo na različnih jedrih.

Poleg skupnega pomnilnika imamo na voljo še prekinitve, s katerimi si lahko jedri med seboj sporočata spremembe v skupnem pomnilniku. Tipično gre za scenarij, ko program, ki teče na enem jedru, vpiše nove podatke v skupni pomnilnik in nato signalizira nasprotnemu jedru, da so novi podatki

na voljo. To pomembno pohitri in poenostavi komunikacijo, kajti v nasprotnem primeru bi se na vsakem jedru morala izvajati programska koda, ki periodično preverja spremembe v skupnem pomnilniku. Prekinitev na nasprotnem jedru sprožimo tako, da opravimo vpis v ustrezni register. Ta vpis nato sproži signal, ki je povezan na vhod prekinitvenega krmilnika nasprotnega jedra. To dogajanje ponazarja tudi slika 3.8, ki prikazuje povezave med jedri in prekinitvenimi krmilniki. Glavno jedro (Cortex-M4) v vpisom v konfiguracijski register M4TXT sproži signal, ki potuje do vhoda prekinitvenega krmilnika na pomožnem jedru (Cortex-M0). Pomožno jedro na podoben način z vpisom v register M0APPTXEVENT sproži prekinitev na glavnem jedru.

3.2.2 Opis protokola za komunikacijo med jedri

Pregled mikrokrmilnika LPC4350 bomo zaključili s primerom protokola, ki ga proizvajalec čipa navaja v uporabniškem priročniku. [11]. Primer vključuje uporabo skupnega pomnilnika SRAM in prej opisanih prekinitev, kot programsko strukturo za prenašanje sporočil pa uporablja krožni medpomnilnik. To je skladno z notranjo podatkovno strukturo, ki jo uporabljamo pri ceveh in sporočilnih vrstah, kar po eni strani dodatno upravičuje našo odločitev o načinu izvedbe medprocesne komunikacije. Vseeno pa ima protokol določene pomanjkljivosti, ki jih bomo analizirali, v poglavju 4 pa opisali rešitev, ki te pomanjkljivosti odpravlja.

Protokol privzema, da imata jedri vnaprej določeni vlogi. Glavno jedro izstavlja ukaze pomožnemu jedru. To prejete ukaze procesira in kot rezultat procesiranja vrača sporočila glavnemu jedru. Pomožno jedro lahko pošlje sporočilo tudi na lastno pobudo. Takšnemu sporočilu pravimo zahteva po strežbi in njegov namen je sporočiti glavnemu jedru, da je pomožno jedro pripravljeno za nov ukaz.

Ker potrebujemo promet v obeh smereh, protokol predvideva uporabo dveh vrst, eno za dostavljanje ukazov proti pomožnemu jedru in drugo za povratna sporočila v smeri proti glavnemu jedru. Vsaka od vrst je zgrajena

(a) Zgradba ukaza

(b) Zgradba sporočila

Slika 3.9: Zgradba ukazov protokola za medprocesno komunikacijo

kot svoj krožni medpomnilnik, za vsak krožni medpomnilnik pa moramo vzdrževati štiri reference. S temi referencami hranimo začetni in končni naslov medpomnilnika ter trenutni položaj branja (označimo jo kot R_r in pisanja (oznaka R_w). Za razlikovanje med praznim in polnim medpomnilnikom vsakič zagotovimo, da je v medpomnilniku vsaj en prazen prostor. Tako torej uporabimo drugačen pristop kot pri sporočilnih vrstah na sistemu FreeRTOS, kjer imamo števec zasedenih mest (stran 47). Pomnilniško lokacijo za shranjevanje referenc definiramo sami, vendar je za zagotovitev sinhronosti sprememb medpomnilnika in registrov pomembno, da se nahaja v istem pomnilniku SRAM, kjer je krožni medpomnilnik.

Glavno jedro dodaja nove podatke v vrsto za ukaze, pomožno jedro pa jih iz nje jemlje. Ob vsakem dodajanju glavno jedro popravlja referenco R_w, pomožno jedro pa R_r. Pri vrsti za sporočila je situacija ravno obrnjena. Tu pomožno jedro dodaja nova sporočila, glavno jedro pa jih jemlje iz vrste. Ob vsakem dodajanju sporočila pomožno jedro popravlja referenco R_w, glavno jedro pa R_r. Po opravljenem dodajanju v vrsto za ukaze oziroma v vrsto za sporočila, se sproži prekinitev po prej opisanem mehanizmu, ki ga prikazuje slika 3.8. Prekinitev sproži tisto jedro, ki je vpisalo podatek, proti jedru, ki je prejemnik.

Slika 3.9 (a) prikazuje zgradbo ukazov. Zgornjih 16 bitov identificira opravilo (proces) na pomožnem jedru, ki mu je ukaz namenjen. Temu sledi 12-bitni naslov, s katerim identificiramo ciljno besedo znotraj procesa. S

Naziv	C	Ima operand
CMD_RD	0	ne
CMD_WR	1	da

Tabela 3.1: Seznam ukazov

Naziv	M	Ima operand	Opis
MSG_SRV	0	ne	Zahteva po strežbi
MSG_RD	1 (branje uspešno)	da	Prebrana beseda
MSG_RD_STS	2 (neveljaven argument) 3 (rezervirano) 4 (rezervirano)	ne	Branje neuspešno, koda napake
MSG_WR_STS	5 (pisanje uspešno) 6 (napaka pri pisanju) 7 (rezervirano)	ne	Pisanje končano, statusna koda

Tabela 3.2: Seznam sporočil

takšnim naslavljanjem lahko naslovimo $2^{12} = 4096$ 32-bitnih pomnilniških besed. Spodnji 4 biti določajo vrsto ukaza (označimo jih kot C), kar skupaj tvori $16 + 12 + 4 = 32$ bitov. Tem 32 bitom lahko sledi še 32-bitni operand, odvisno od vrste ukaza. V tabeli 3.1 so našteti ukazi, kot jih definira protokol, skupaj z vrednostjo polja C in navedbo prisotnosti operanda.

Zgradba sporočil je podobna zgradbi ukazov in je prikazana na sliki 3.9 (b). Ponovno imamo zgornjih 16 bitov kot identifikator opravila na pomožnem jedru, ki tokrat sporoča, od kod sporočilo izvira. Spodnji 4 biti (torej biti 3-0, označimo jih kot M) določajo vrsto sporočila in jih podobno kot pri ukazih skupaj s prisotnostjo opcijskega operanda razlaga tabela 3.2. Tudi tem bitom lahko glede na vrsto sporočila sledi 32-bitni operand.

Uporaba bitov 15-4 se razlikuje glede na vrsto sporočila. Če je sporočilo vrste MSG_SRV, uporabimo bite 15-8 (skupaj 8 bitov) za opis vrste storitve,

spodnje 4 bite (biti 7-4) pa postavimo na 0. Nabor vrednosti za vrste storitev je poljuben in prepuščen uporabniku, da jih definira glede na lastne potrebe. Za vse druge vrste sporočil uporabimo vseh 12 bitov (torej biti 15-4), v katerih shranimo isti naslov, ki smo ga prejeli pri ukazu za branje oziroma pisanje.

Če opisani protokol primerjamo z bolj splošnimi mehanizmi medprocesne komunikacije, ki smo jih predstavili v poglavju 2, lahko ugotovimo nekaj pomanjkljivosti, predvsem na račun splošnosti in preprostosti uporabe. Kot prvo sta vlogi glavnega in pomožnega jedra vnaprej predvideni, pri čemer eno jedro izstavlja ukaze, drugo pa jih izvršuje. Ti dve vlogi lahko sicer obrnemo, a s tem se še vedno ne znebimo asimetričnosti, ki nam vnaprej vsiljuje način delovanja dveh opravil. Protokol sicer ponuja dober predlog oblike sporočil za primer, ko potrebujemo izstavljanje ukazov in vračanje rezultatov njihovega izvajanja. Vendar pa lahko ta sporočila preprosto preslikamo na podatkovne strukture v višjenivojskem programskem jeziku, ki jih nato pošiljamo preko bolj splošnega vmesnika.

Naslednja slabost je, da se na obeh jedrih lahko izvaja le po eno opravilo, ki dostopa do mehanizma (krožnega medpomnilnika za sporočila in registrov). V nasprotnem primeru moramo sami poskrbeti za usklajenost oziroma atomarnost dostopov. Bolj formalno bi lahko rekli, da moramo zadostiti dodatnim precedenčnim omejitvam oblike:

$$cmd_i \preceq cmd_{i+1}$$

in

$$msg_i \preceq msg_{i+1}$$

kjer s cmd_i označujemo i-ti posel pošiljanja ukaza v nekem urejenem zaporedju poslov $\langle cmd_1, cmd_2, ..., cmd_m \rangle$ in z msg_i i-ti posel pošiljanja sporočila v nekem urejenem zaporedju poslov $\langle msg_1, msg_2, ..., msg_n \rangle$.

Dodajmo še, da je protokol definiran na dokaj nizkem nivoju, saj je že popravljanje referenc prepuščeno uporabniku. Vse to dodatno zaplete upo-

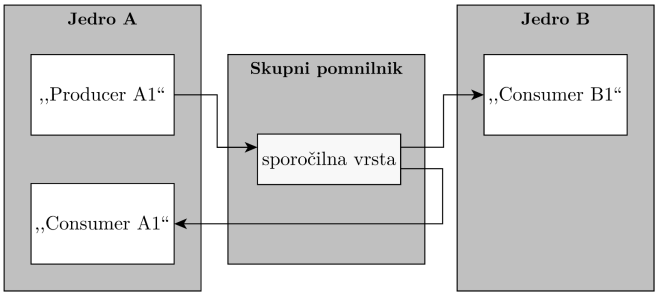
rabo in povzroča večjo dovzetnost do napak pri izvedbi na strani uporabnika. Rešitev, ki jo bomo predstavili v poglavju 4, odpravlja vse naštete slabosti, hkrati pa združuje prednosti mehanizma sporočilnih vrst z učinkovito rabo skupnega pomnilnika, kot ga prikazuje pravkar opisani protokol.

Poglavje 4

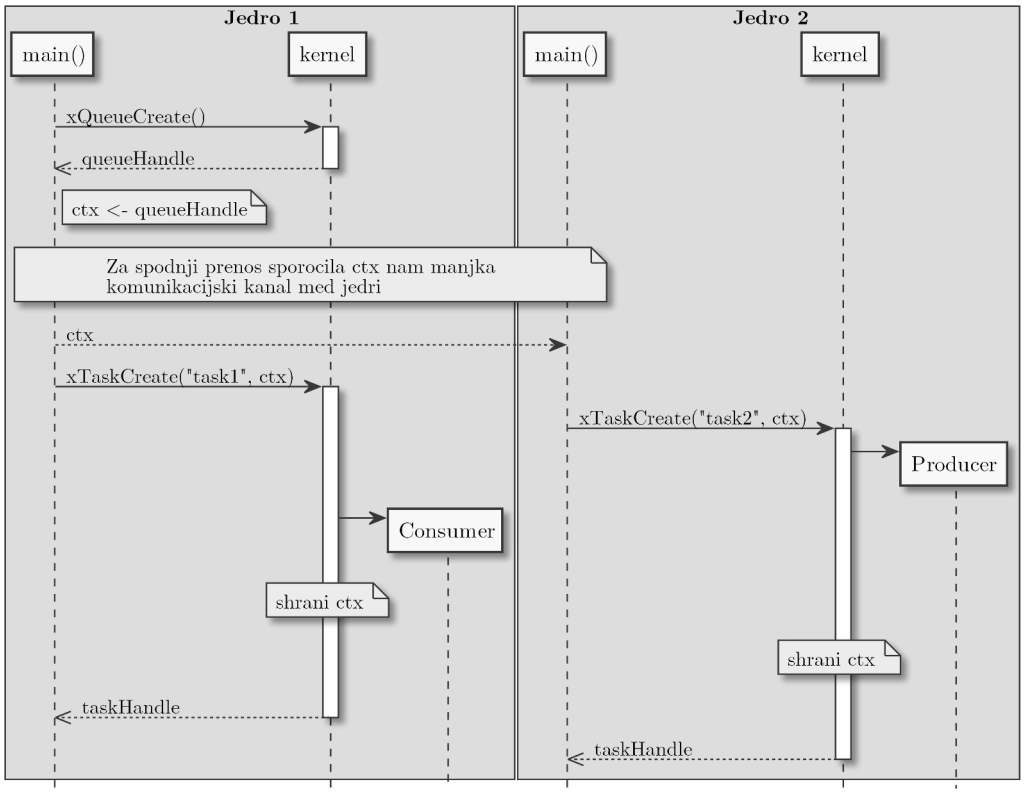
Izvedba

V tem poglavju bomo predstavili izvedbo sporočilnih vrst za operacijski sistem FreeRTOS, ki jih lahko uporabimo na dvojedrnem procesorju LPC4350. Za komunikacijo med jedri bomo po vzoru na protokol proizvajalca (razdelek 3.2, Mikrokrmilnik LPC4350) uporabili skupni pomnilnik in prekinitve. Čeprav bomo naredili povsem novo izvedbo, brez ponovne uporabe obstoječe programske kode sporočilnih vrst na FreeRTOS, bomo poskušali čim bolj ohraniti podobnost z njihovim programskim vmesnikom. S takšnim pristopom upamo, da bomo olajšali prenos že obstoječih programskih rešitev na dvoprocesorski sistem.

Našo sporočilno vrsto bo možno uporabljati z večjim številom proizva-



Slika 4.1: Položaj sporočilne vrste v sistemu

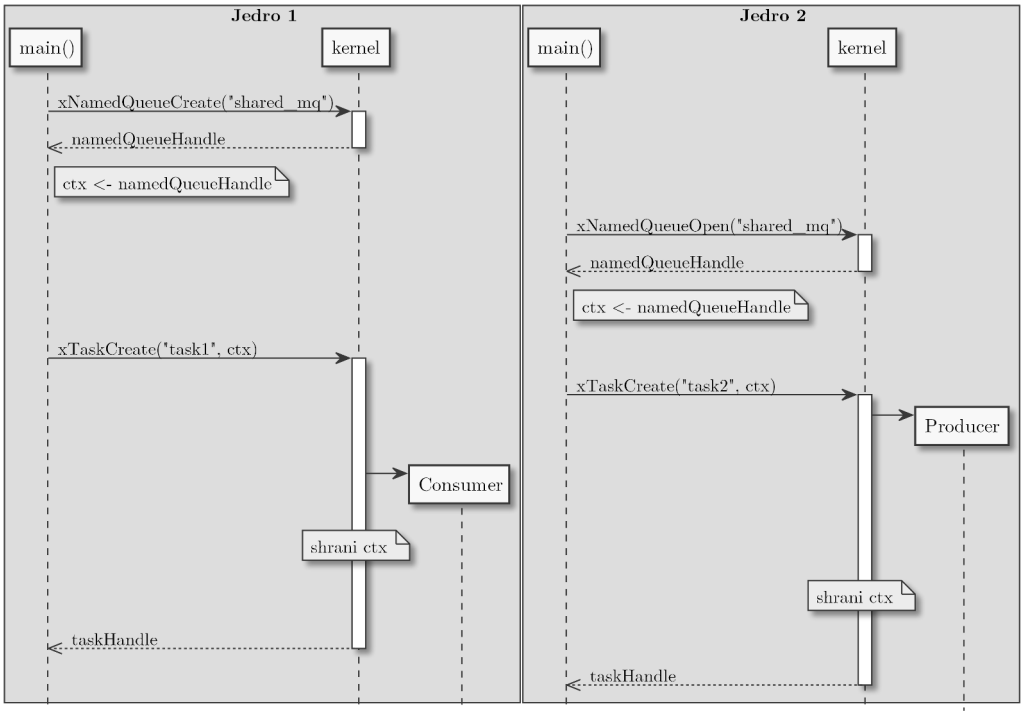


Slika 4.2: Prikaz problema prenosa programske ročice med jedri

jalcev in porabnikov, ki se lahko izvajajo na poljubnem jedru. Slika 4.1 prikazuje položaj vrste v sistemu in je obenem primer, ko dva porabnika na različnih jedrih jemljeta sporočila iz iste vrste.

4.1 Opis programskega vmesnika

Programski vmesnik sporočilne vrste za večjedrni sistem, kot ga lahko vidimo v izseku kode 4.1, se razlikuje le malo glede na svoj izvirnik na strani 40. Glavna sprememba je v načinu, kako sporočilno vrsto ustvarimo oziroma jo odpremo. V izvirnem vmesniku imamo zgolj funkcijo za ustvarjanje vrste `xQueueCreate`. Eksplicitno odpiranje vrste ni potrebno. Namesto tega si shranimo njeno programsko ročico in jo podajamo drugim opravilom, kot smo to



Slika 4.3: Pridobitev programske ročice na različnih jedrih

tudi prikazali v razdelku 3.1.1.

Zakaj je takšen način podajanja programske ročice težaven v dvojedrnem sistemu, lahko vidimo s pomočjo diagrama na sliki 4.2. Tu izhajamo iz zasnove, ki je značilna asimetričnim večjedrnim sistemom in kjer na vsakem procesorskem jedru teče ločen primerek operacijskega sistema. Znotraj posameznega jedra lahko vidimo sekvenco izmenjave sporočil med glavno funkcijo main(), jedrom operacijskega sistema (kernel) in enim opravilom. Na jedru 1 je to opravilo porabnik (Consumer), na jedru 2 pa proizvajalec (Producer).

Osnovna ideja je preprosta. Ko znotraj enega primerka operacijskega sistema, v našem primeru na jedru 1, ustvarimo sporočilno vrsto, moramo njeno programsko ročico na nek način dostaviti operacijskem sistemu na jedru 2. Nato lahko na jedru 2 nadaljujemo z ustvarjanjem opravil, ki bodo uporabljala to sporočilno vrsto za komunikacijo z opravili na jedru 1. Težava pri tej ideji je v tem, da med jedri nimamo vzpostavljenega nekega komunikacijskega

kanala, po katerem lahko prenesemo vrednost kontekstne spremenljivke ctx. Prav en takšen kanal pa bi radi ustvarili s sporočilno vrsto.

Na nivoju programskega vmesnika to težavo rešimo s poimenovanjem sporočilnih vrst. Vsaki sporočilni vrsti ob njenem ustvarjanju dodelimo neko enolično ime. Z našim programskim vmesnikom to storimo s funkcijo `xNamedQueueCreate`, katere prvi parameter je ime vrste, naslednja dva pa sta enaka kot pri izvirniku (funkcija `xQueueCreate`). Vsakič, ko potrebujemo programsko ročico do neke vrste, uporabimo posebno programsko funkcijo, ki kot parameter vzame ime vrste, njen rezultat pa je programska ročica do te vrste. Tej operaciji pridobitve programske ročice (oziroma reference) na objekt medprocesne komunikacije bomo skladno z uveljavljeno prakso rekli *odpiranje*. Zato tovrstno funkcijo v našem programskem vmesniku imenujemo `xNamedQueueOpen`. Programsko ročico na jedru 2 tako dobimo na način, kot ga prikazuje slika 4.3.

Rešitev s poimenovanjem objektov medprocesne komunikacije ni nova in jo srečamo tudi na številnih drugih operacijskih sistemih. Takšen primer so denimo imenovane cevi, pri katerih dva nepovezana procesa pridobita programsko ročico cevi[1] tako, da odpreta cev s točno določenim imenom.

Na tem mestu omenimo še eno izboljšavo, ki se je neposredno ne da razbrati iz programskega vmesnika, a programerju olajša uporabo. Izvirni vmesnik vsebuje funkcijo `vQueueDelete`, ki izbriše sporočilno vrsto in sprosti vse z njo uporabljene sistemske vire. Težava pri tej funkciji je, da je smemo uporabiti šele takrat, ko vemo, da vrste več ne bo uporabljalo nobeno drugo opravilo. Pri njeni uporabi moramo zato biti posebej previdni, kar pa lahko zaplete programsko kodo, še posebej, če imamo opravila na različnih jedrih, ki si morajo predhodno izmenjati informacijo o tem, da vrste več ne bodo uporabljala.

Funkcija `vNamedQueueClose` olajša način sproščanja vrste tako, da v paru s funkcijo `xNamedQueueOpen` vodi števec referenc. Ko vrsto ustvarimo, je ta števec enak 1, nato pa se z vsakim novim klicem funkcije `xNamedQueueOpen` poveča za

[1] običajno v obliki datotečnega opisovalnika

1 oziroma z vsakim klicem `vNamedQueueClose` zmanjša za 1. Če pri tem števec doseže vrednost 0, se opravi tudi sprostitev vseh z vrsto zasedenih sistemskih virov. Posamično opravilo torej s funkcijo `vNamedQueueClose` naznani, da vrste več ne bo uporabilo. Ko to naznani še zadnje opravilo, se v ozadju izvede koda za sprostitev sistemskih virov. Potrebe po eksplicitnem brisanju vrste več ni, zato v našem programskem vmesniku ni moč najti ekvivalenta funkciji `vQueueDelete`.

Sporočilnim vrstam smo dodali še eno priročno funkcionalnost, ki smo jo že omenili v poglavju 2 na strani 27 in jo pozna tudi standard POSIX.1-2001. To je dodelitev numerične prioritete poslanim sporočilom, ki lahko zavzame eno izmed petih vrednosti in so naštete v naštevanju `enum ItemPriority_t`. Prioriteto podamo preko parametra `xPriority` pri funkcijah za pošiljanje (`xNamedQueueSendToBackPrio` oziroma `xNamedQueueSendToFrontPrio`, ki jo uporabljamo za način LIFO).

Kot zadnje omenimo še dodano predpono v imenih funkcij. Ker imamo opravka z novo izvedbo sporočilnih vrst, ki ne izključuje že obstoječe, smo v izogib konfliktu v imenskem prostoru prisiljeni v vpeljavo novih imen. Našo sporočilno vrsto bomo zato poimenovali *imenovana sporočilna vrsta* oziroma skrajšano kar *imenovana vrsta*[2]. Čeprav s skrajšanim imenom izpuščamo pomemben atribut, si bomo zaradi praktičnosti krajših imen funkcij dovolili takšno ohlapnost. Nenazadnje jo lahko zasledimo tudi v izvirnem programskem vmesniku sistema FreeRTOS.

Izsek kode 4.1: Programski vmesnik sporočilne vrste za večjedrni sistem

```
1 enum ItemPriority_t
2 {
3     PRI_LOWEST = 0,
4     PRI_LOW,
5     PRI_NORMAL,
6     PRI_HIGH,
7     PRI_HIGHEST,
```

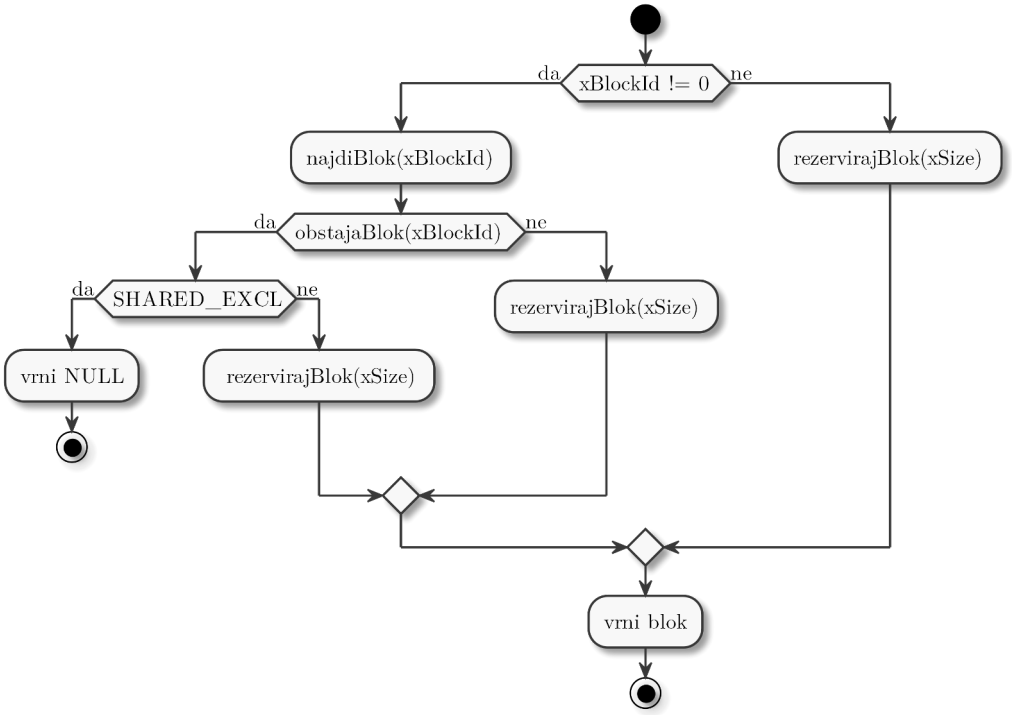
[2]S takšnim poimenovanjem se zgledujemo po imenovanih ceveh, ki jih srečamo v številnih operacijskih sistemih in tudi v standardu POSIX.1-2001.

```
8
9       ItemPriorityCount
10  };
11
12  NamedQueueHandle_t xNamedQueueCreate( const char *pcName,
13      UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
14
15  NamedQueueHandle_t xNamedQueueOpen( const char *pcName );
16
17  void vNamedQueueClose( NamedQueueHandle_t xNamedQueue );
18
19  BaseType_t xNamedQueueSendToBackPrio( NamedQueueHandle_t xNamedQueue,
20      const void *pvItemToQueue, TickType_t xTicksToWait,
21      UBaseType_t xPriority );
22
23  BaseType_t xNamedQueueSendToFrontPrio( NamedQueueHandle_t xNamedQueue,
24      const void *pvItemToQueue, TickType_t xTicksToWait,
25      UBaseType_t xPriority );
26
27  BaseType_t xNamedQueueReceive( NamedQueueHandle_t xNamedQueue,
28      void *pvBuffer, TickType_t xTicksToWait );
29
30  BaseType_t xNamedQueuePeek( NamedQueueHandle_t xNamedQueue,
31      void *pvBuffer, TickType_t xTicksToWait );
```

4.2 Izvedba sporočilne vrste

4.2.1 Uporabljeni programski moduli

Pri opisu izvedbe sporočilne vrste bomo privzeli obstoj nekaterih dodatnih programskih modulov, ki jih potrebujemo za izvedbo naše sporočilne vrste. Na tej točki jih bomo opisali le iz funkcionalnega vidika, medtem ko bomo njihovo podrobno izvedbo predstavili kasneje.



Slika 4.4: Rezervacija bloka v skupnem pomnilniku

Skupni pomnilnik

Na voljo imamo skupni pomnilnik z operacijo `pvSharedMalloc`, ki v segmentu glavnega pomnilnika, označenega za skupno rabo med procesorskimi jedri, rezervira blok želene velikosti in vrne njegov pomnilniški naslov. V primeru neuspele rezervacije funkcija vrne vrednost `NULL`. Podpis programske funkcije `pvSharedMalloc` je sledeč:

```
void *pvSharedMalloc( size_t xSize, uint32_t xBlockId,
                      uint32_t xFlags,
                      portBASE_TYPE *xNew ) PRIVILEGED_FUNCTION;
```

kjer je:

`xSize`: želena velikost bloka v bajtih.

`xBlockId`: enolična celoštevilska oznaka bloka, pri čemer je `0` posebna vrednost, s katero povemo, da blok nima oznake. V skupnem pomnilniku imamo lahko poljubno mnogo blokov brez oznake in zato je vrednost `0` v smislu enoličnosti izjema.

`xFlags`: parameter za podajanje različnih zastavic. Za naše potrebe definiramo le eno zastavico z oznako `SHARED_EXCL` in jo bomo opisali v nadaljevanju.

`xNew`: izhodni parameter, s katerim funkcija sporoči, ali vrnjeni naslov pripada na novo rezerviranemu bloku (vrednost `1`) ali že obstoječemu (vrednost `0`). Če je `xBlockId` enak 0, potem ta parameter nima posebnega pomena, saj se bloki brez oznake vedno na novo rezervirajo.

Posebnost funkcije `pvSharedMalloc` je v tem, da lahko rezerviranim blokom dodeli enolično oznako. Kako ta oznaka vpliva na postopek rezervacije, povemo z zastavico `SHARED_EXCL`.

- Če je zastavica `SHARED_EXCL` postavljena, potem bo rezervacija neuspešna v kolikor v skupnem pomnilniku že obstaja blok z oznako, ki smo jo podali s parametrom `xBlockId`. To seveda ne velja za primer, ko je `xBlockId` enak `0`, saj s tem povemo, da blok nima oznake. Rezervacija bloka brez oznake je vedno uspešna, razen če je v pomnilniku premalo prostora za blok želene velikosti.

- Če zastavice `SHARED_EXCL` ni, potem bo funkcija `pvSharedMalloc` rezervirala nov blok, v kolikor v skupnem pomnilniku še ni bloka z oznako `xBlockId`. V tem primeru bo tudi `xNew` postavila na vrednost `1`. Če takšen blok že obstaja in je njegova velikost enaka ali večja kot `xSize`, potem bo funkcija vrnila naslov tega bloka in `xNew` bo enak `0`. V nasprotnem primeru, ko je velikost obstoječega bloka manjša kot `xSize`, vrne vrednost `NULL`. Če kot `xBlockId` podamo vrednost `0`, velja isto kot za prejšnji primer, ko je zastavica postavljena.

Zaradi lažjega razumevanja je pravkar opisani postopek prikazan z diagramom na sliki 4.4.

Mutex

Na voljo imamo mutex, ki ga lahko uporabimo v dvoprocesorskem sistemu. Vstopni protokol je realiziran s funkcijo `vMutexLock`, izstopni pa z `vMutexUnlock`.

Dodatno imamo na voljo še funkcijo `vMutexTryLock`, pri kateri je vstopni protokol spremenjen tako, da ne blokira izvajanja. Če pogoj za vstop v kritično območje ni izpolnjen, funkcija namesto da bi čakala oziroma ponovno preizkusila vstopni pogoj, takoj vrne napako.

Pred prvo uporabo moramo mutex inicializirati s funkcijo `vMutexInit`.

4.2.2 Predstavitev sporočilne vrste v pomnilniku

Izbira podatkovne strukture za predstavitev sporočilne vrste

Uporabo krožnega medpomnilnika smo kot strukturo za sporočilno vrsto doslej srečali na dveh mestih. Prvič pri pregledu sporočilnih vrst na sistemu FreeRTOS, drugič pa pri obravnavi protokola za izmenjavo sporočil med jedri, ki ga navaja proizvajalec mikrokrmilnika. Krožni medpomnilnik se nam zato ponuja kot logična izbira za našo izvedbo. Vseeno pa moramo premisliti, ali je še vedno primeren za uporabo ob vseh dodatnih zahtevah, ki smo jih navedli v uvodnem delu tega poglavja. Zaradi boljše predstave strnimo te zahteve z naslednjim seznamom:

- uporaba v dvojedrnem sistemu,
- uporaba v sistemih v stvarnem času,
- štetje referenc,
- prioritete sporočil.

Najprej analizirajmo prvo zahtevo. Če obdržimo krožni medpomnilnik, ga bomo morali hraniti v skupnem pomnilniku, kjer bo dostopen iz vseh

jeder, ki sporočilno vrsto uporabljajo. Nato bomo morali v funkcijah nad krožnim medpomnilnikom identificirati vsa kritična območja in jih zaščititi z neko primerno izvedbo mutex-a (torej takšno, ki jo lahko uporabimo na dvoprocesorskem sistemu). Doslej se gibljemo znotraj začrtanih okvirjev, saj smo uporabo programskih modulov skupnega pomnilnika in mutex-a že predvideli. Da bi takšna sporočilna vrsta delovala v dvojedrnem sistemu, moramo le še poskrbeti za signalizacijo nasprotnega jedra ob dodajanju ali odvzemanju sporočila iz medpomnilnika. Pri tem se lahko zgledujemo po principu iz protokola proizvajalca, kjer je to doseženo s pomočjo prekinitev. Trenutno torej ne kaže, da bi se morali krožnemu medpomnilniku odreči.

Ker je krožni medpomnilnik uporabljen že v izvirni izvedbi sporočilnih vrst operacijskega sistema v stvarnem času (FreeRTOS), lahko upravičeno domnevamo, da je iz vidika sistemov v stvarnem času njegova uporaba smotrna. Če preučimo postopke dodajanja sporočil (na začetek ali konec vrste) in sprejemanja še z vidika časovne zahtevnosti, opazimo, da se vse operacije izvedejo v času $O(1)$. Ta lastnost je za nas ugodna in bi jo radi obdržali, saj le še dodatno potrjuje upravičenost rabe krožnega medpomnilnika za delo v stvarnem času.

Štetje referenc zadeva zgolj mehanizem odpiranja in zapiranja sporočilne vrste in ne posega niti v strukturo sporočil niti v način izvajanja operacij nad njimi (pošiljanje in prejemanje).

Težave se pojavijo pri zadnji zahtevi, kjer moramo pri jemanju sporočil iz vrste upoštevati njihovo prioriteto. S krožnim medpomnilnikom na nek preprost način tega ne moremo zagotoviti, sploh če želimo ohraniti časovno zahtevnost $O(1)$. Na tem mestu se nam morda utrne ideja, da bi uporabili prioritetno vrsto, a tudi tam časovna zahtevnost ni tako dobra, saj je za dodajanje in odvzemanje enaka $O(\lg n)$ (pri čemer je n število sporočil v vrsti). [17]

Očitno potrebujemo nek drug pristop. Pomagajmo si z dejstvom, da imamo fiksno število celoštevilskih prioritet, ki tvorijo neko zaporedje. V našem primeru so to števila 0...4. Če bi lahko za vsako izmed prioritet

vzdrževali svojo vrsto v obliki krožnega medpomnilnika, bi se približali našemu cilju. Vendar se pojavi težava s prostorom, kajti pri `ItemPriorityCount` prioritetah sedaj rabimo `ItemPriorityCount`-krat toliko pomnilnika kot v izvedbi z enim samim krožnim pomnilnikom. Predvideti moramo najbolj neugodno situacijo, ko imamo v vrsti maksimalno možno sporočil, vsa z enako prioriteto. Ta prioriteta pa je lahko katerakoli izmed `ItemPriorityCount` možnih.

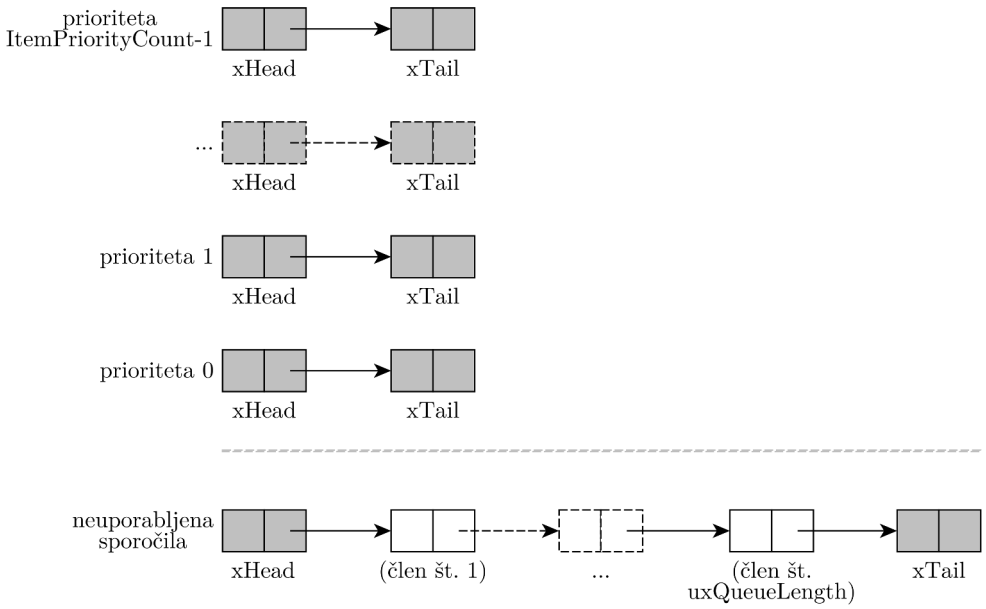
Vse kaže, da potrebujemo neko prožnejšo strukturo, s katero lahko zgradimo vrsto poljubne dolžine, ne da bi ji namenili nek fiksen del pomnilnika. Namesto krožnega medpomnilnika zato uporabimo povezan seznam. Tako dobimo `ItemPriorityCount` povezanih seznamov, ki so na začetku prazni, tem pa dodajmo še enega, v katerem hranimo vsa preostala, še neuporabljena sporočila (slika 4.5). Na začetku, ko je vrsta še prazna, je teh `uxQueueLength`. Ta seznam neuporabljenih sporočil uporabimo za vnaprejšnjo rezervacijo vseh možnih členov, ki jih nato zgolj premikamo med seznami. Ko prispe novo sporočilo s prioriteto p, vzamemo prvi prosti člen iz vrste neuporabljenih sporočil in ga dodamo v seznam sporočil prioritete p. Ob jemanju sporočila s prioriteto p naredimo ravno obratno: vzamemo prvi člen iz seznama sporočil prioritete p in ga dodamo nazaj v seznam neuporabljenih sporočil.

Predstavitev sporočilne vrste s povezanimi seznami

Podatkovno strukturo, s katero predstavimo posamezno sporočilo (na sliki 4.5 prikazano v obliki dvodelne škatle) imenujemo `struct QueueItem_t` in je prikazana v izseku kode 4.2. Član `pxNext` je referenca na naslednje sporočilo in je na sliki predstavljen z desno polovico škatle. Član `pucData` služi kot referenca na vsebino sporočila, katere maksimalna dolžina je lahko `uxItemSize` bajtov in je na sliki predstavljen z levo polovico škatle.

Izsek kode 4.2: Člen v povezanem seznamu sporočil

```
struct QueueItem_t
{
    struct QueueItem_t *pxNext;
    uint8_t *pucData;
```



Slika 4.5: Začetna postavitev v sporočilni vrsti s povezanimi seznami

```
};
```

V nadaljevanju si oglejmo, kako člene tipa `struct QueueItem_t` dodajamo in jemljemo iz seznama. Izsek kode 7.2 (Dodatek, stran 106) prikazuje vse funkcije, ki jih uporabljamo v zvezi s povezanim seznamom.

Začnimo z razlago podatkovne strukture, s katero definiramo seznam (`struct ItemList_t`, vrstice 1-6). Seznam se začne in zaključi s posebnim členom, čigar vloga je zgolj označevanje začetka oziroma konca seznama. Takemu členu pravimo *stražar* in nam pomaga poenostaviti robne pogoje v programski kodi za dodajanje in odvzemanje členov iz seznama. Oba stražarja pripravimo že v sami strukturi in ju imenujemo `xHead` (stražar na začetku) in `xTail` (stražar na koncu). Dodatno uvedemo še referenco na zadnji člen v seznamu `pxLast`, ki je pred stražarjem na koncu. Tako `pxLast` vedno kaže na zadnji člen, ki smo ga dodali v seznam, razen v primeru praznega seznama, ko kaže na `xHead`. Referenco `pxLast` bomo uporabili za hitro dodajanje členov

na konec seznama.

Seznam pripravimo za uporabo s funkcijo `itemListInit`. Po izvedbi te funkcije je seznam, ki smo ga podali preko parametra `pxItemList`, veljaven in prazen. Funkcija poveže člena `xHead` in `xTail` tako, da član `pxNext` člena `xHead` kaže na `xTail` (vrstica 13), s čimer definiramo prazen seznam. To lahko vidimo tudi na sliki 4.5, kjer so vsi seznami razen zadnjega prazni. Pri funkciji `itemListIsEmpty` to lastnost izkoristimo za ugotavljanje praznega seznama (vrstice 22 in 8-9). Ker je `xTail` zadnji člen v seznamu, zanj postavimo `pxNext` na vrednost `NULL` (vrstica 14). Pri stražarjih nikdar ne uporabljamo reference `pucData`, zato ji priredimo kar vrednost `NULL` (vrstica 15). Na koncu še definiramo `pxLast` tako, da kaže na `xHead`, kar je skladno s tem, kar smo v prejšnjem odstavku povedali za prazen seznam.

Funkcija `itemListFirst` vrne prvi člen v seznamu, ki ga pridobi s pomočjo stražarja `xHead` (vrstica 31). Če je seznam prazen, funkcija vrne vrednost `NULL`.

Funkcija `itemListPrepend` doda nov člen `pxItem` na začetek seznama tako, da ga vrine med stražarjem `xHead` in členom, ki stražarju sledi. Zato v vrstici 48 povežemo `pxItem` s členom, ki sledi stražarju `xHead`, stražarja `xHead` pa s členom `pxItem` (vrstica 49). Če dodajamo v prazen seznam, moramo popraviti še referenco `pxLast` tako, da kaže na dodani člen `pxItem` (vrstice 43-46). V nasprotnem primeru popravljanje `pxLast` ni potrebno, saj dodajamo na začetek seznama in bo zato zadnji člen v seznamu še vedno isti.

Funkcija `itemListAppend` doda nov člen `pxItem` na konec seznama tako, da ga vrine med zadnjim členom v seznamu, ki ga dobimo z referenco `pxLast`, in stražarjem na koncu seznama `xTail`. V programski kodi to opravimo v vrsticah 55-56, kjer `pxItem` povežemo s stražarjem `xTail`, člen, ki ga dobimo s `pxLast`, pa s `pxItem`. Ker dodajamo na konec seznama, moramo za razliko od funkcije `itemListPrepend` vedno popraviti referenco `pxLast` (vrstica 57), ki naj kaže na pravkar dodani člen `pxItem`.

Nazadnje si oglejmo še funkcijo `itemListRemove`, ki odstrani prvi člen iz seznama in ga vrne kot rezultat. Podobno kot pri funkciji `itemListFirst`,

pridobimo prvi člen s pomočjo stražarja `xHead` in ga shranimo kot rezultat funkcije `pxResult` (vrstica 68). Nato v vrstici 69 povežemo stražarja `xHead` s členom, ki sledi prvemu členu (to je naslednji od `pxResult`). Ker člen `pxResult` odvzemamo iz seznama, prekinemo povezavo med njim in naslednjim členom v seznamu tako, da njegovemu članu `pxNext` priredimo vrednost `NULL` (vrstica 70). Dodajmo še to, da v primeru praznega seznama funkcija `itemListRemove` vrne vrednost `NULL` (to zagotovimo z vnaprejšnjo priredbo `pxResult = NULL` v vrstici 64 in s pogojnim stavkom v vrstici 66).

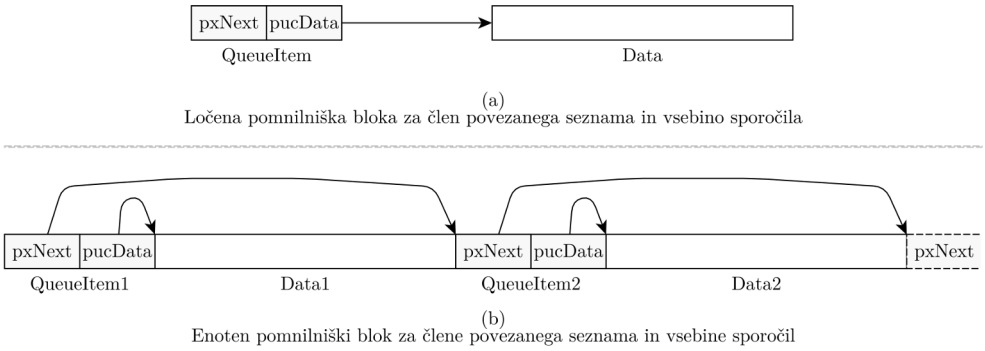
Po pregledu naštetih funkcij lahko brez posebnih težav ugotovimo, da se vse izvedejo v času $O(1)$, kar pomeni, da smo na dobri poti, da krožni medpomnilnik zamenjamo s povezanim seznamom.

Predstavitev celotne sporočilne vrste

Izsek kode 4.3 prikazuje podpis podatkovne strukture `struct NamedQueue_t`, s katero predstavljamo sporočilno vrsto v njeni celoti:

Izsek kode 4.3: Programska struktura za sporočilno vrsto

```
struct NamedQueue_t
{
    char cName[ QUEUE_NAME_MAX ];

    UBaseType_t uxQueueLength;
    UBaseType_t uxItemSize;

    UBaseType_t uxReferenceCount;

    portMUTEX_TYPE xMutex;

    UBaseType_t uxIndex;

    struct ItemList_t xUsedItems[ ItemPriorityCount ];
    struct ItemList_t xFreeItems;

    uint8_t ucItemBuffer[];
```



Slika 4.6: Različna pristopa k rezervaciji pomnilnika za sporočila

```
};
```

Član `pxUsedItems` ustreza `ItemPriorityCount` seznamom uporabljenih sporočil, `xFreeItems` pa seznamu neuporabljenih sporočil (slika 4.5). Vse sezname predstavimo s programsko strukturo `struct ItemList_t`, ki smo jo skupaj z vsemi potrebnimi operacijami že spoznali.

Vrnimo se na kratko k opisu členov v povezanem seznamu (struktura `struct QueueItem_t`). Člen je sestavljen iz dveh referenc: ene na naslednji člen v seznamu in druge na vsebino sporočila. V primerih, ko referenco na podatke pridobimo s strani zunanjega uporabnika, imamo dva ločena bloka pomnilnika, kot to prikazuje slika 4.6 a. Prvega rezerviramo za hrambo samega člena (oznaka QueueItem), drugega pa je rezerviral uporabnik za vsebino (oznaka Data). Če se držimo tega pristopa, bomo morali med uporabo sporočilne vrste opraviti veliko število majhnih rezervacij, po dve za vsako sporočilo. To je lahko časovno precej potratno in tudi z vidika fragmentacije pomnilnika ima lahko zelo slab učinek (spomnimo se prikaza fragmentacije pomnilnika na strani 46). Zato bomo raje vnaprej rezervirali en sam blok, ki bo po velikosti zadoščal za hrambo vseh členov povezanega seznama in vsebine vseh sporočil. Notranjo delitev bloka na člene povezanega seznama in bloke z vsebino sporočil bomo opravili naknadno po rezervaciji na način, kot ga prikazuje slika 4.6 b.

Najprej izračunajmo, kako velik naj bo rezerviran blok. Vemo, da je maksimalna dolžina posameznega sporočila enaka `uxItemSize` bajtov. To število moramo zaokrožiti navzgor glede na velikost, ki je predpisana s strojno arhitekturo in izvira iz zahteve, da morajo biti dostopi do pomnilnika poravnani. Točneje to pomeni, da morajo biti pomnilniški naslovi, ki jih tvori procesor, večkratniki nekega števila[3] (običajno pomnilniške besede), ki ga označimo kot A. Ker bomo do vsakega sporočila znotraj rezerviranega bloka dostopali preko svojega lastnega naslova, shranjenega v članu `pucData` (struktura `struct QueueItem_t`), moramo poskrbeti, da bo vsak tak naslov poravnan. To bomo zagotovili tako, da bomo vsakemu sporočilu namenili takšno količino pomnilnika, ki je večja ali enaka `uxItemSize` in je najmanjši takšen večkratnik števila A. Dolžino posameznega sporočila v pomnilniku zato izračunamo po naslednji enačbi:

$$L = S + A - 1 - ((S + A - 1) \mod A) \tag{4.1}$$

kjer je S dolžina sporočila v bajtih, A število bajtov, na katere moramo poravnati pomnilniške dostope, in L navzgor zaokrožena dolžina sporočila v bajtih. Enačbo 4.1 zapišemo v izvorni kodi kot makro za izračun poravnane velikosti `ALIGNED_SIZE` (izsek kode 4.4, vrstice 1-4), pri čemer A zamenjamo s sistemsko določeno količino `portBYTE_ALIGNMENT`.

Da bi dobili velikost bloka, ki ga sporočilo zasede skupaj s pripadajočim členom v povezanem seznamu, uporabimo makro `NQUEUE_ITEM_SIZE` (izsek kode 4.4, vrstice 6-9), s katerim dobimo vsoto poravnane velikosti `uxItemSize` in poravnane velikosti strukture `struct QueueItem_t`. Če to vsoto zmnožimo z maksimalnim številom sporočil v vrsti `uxQueueLength`, dobimo velikost bloka, ki ga potrebujemo za hrambo vseh sporočil. Ta izračun opravimo s pomočjo makra `NQUEUE_TOTAL_ITEMS_SIZE` (vrstice 11-13).

Izsek kode 4.4: Izračun velikosti poravnanih programskih struktur

```
1 #define ALIGNED_SIZE( S ) (                                  \
2     ( S + portBYTE_ALIGNMENT - 1 ) -                         \
```

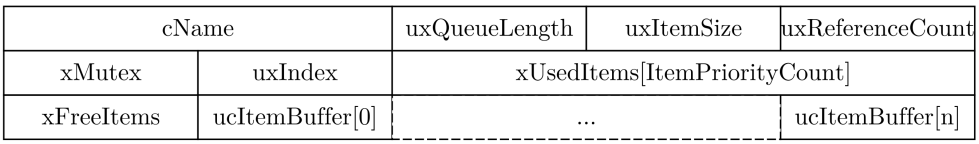
[3]Na sistemu FreeRTOS to število določimo s simbolom `portBYTE_ALIGNMENT`.

```
3     ( ( S + portBYTE_ALIGNMENT - 1 ) % portBYTE_ALIGNMENT ) \
4 )
5
6 #define NQUEUE_ITEM_SIZE( uxItemSize ) (         \
7     ALIGNED_SIZE( uxItemSize ) +                 \
8     ALIGNED_SIZE( sizeof( struct QueueItem_t ) ) \
9 )
10
11 #define NQUEUE_TOTAL_ITEMS_SIZE( uxItemSize, uxQueueLength ) ( \
12     NQUEUE_ITEM_SIZE( uxItemSize ) * uxQueueLength             \
13 )
14
15 #define NQUEUE_SIZE( uxItemSize, uxQueueLength ) (       \
16     sizeof( struct NamedQueue_t ) +                      \
17     NQUEUE_TOTAL_ITEMS_SIZE( uxItemSize, uxQueueLength ) \
18 )
```

Sedaj ko vemo, kako rezervirati dovolj velik blok za vsa sporočila, se bomo lotili gradnje seznama, kot ga prikazuje slika 4.6 b. Ta seznam bomo predstavili s članom `xFreeItems` strukture `struct NamedQueue_t`. Recimo, da smo naslov rezerviranega bloka shranili v spremenljivko `pucBuffer`. Naslednji izsek kode zgradi seznam neuporabljenih sporočil (`xFreeItems` v strukturi `struct NamedQueue_t`):

Izsek kode 4.5: Gradnja seznama neuporabljenih sporočil

```
1  UBaseType_t i;
2  struct QueueItem_t *pxQueueItem;
3
4  for( UBaseType_t i = 0; i < uxQueueLength; ++i )
5  {
6      pxQueueItem = ( struct QueueItem_t * ) pucBuffer;
7      pxQueueItem->pucData =
8          &pucBuffer[ ALIGNED_SIZE( sizeof( struct QueueItem_t ) ) ];
9
10     itemListAppend( &pxNamedQueue->xFreeItems, pxQueueItem );
11
12     pucBuffer += NQUEUE_ITEM_SIZE( uxItemSize );
```



Slika 4.7: Pomnilniški blok z vsebovano sporočilno vrsto

```
13 }
```

V zanki se sprehodimo skozi celotni rezerviran blok po korakih, ki so enaki `NQUEUE_ITEM_SIZE( uxItemSize )` (vrstica 12). Kot smo že prej izračunali, je to navzgor zaokrožena velikost bloka, ki ga zasede eno sporočilo skupaj s pripadajočim členom v povezanem seznamu. V vsaki iteraciji zanke vzamemo trenutni naslov `pucBuffer` kot naslov novega člena povezanega seznama (vrstica 6). Ta člen nato dereferenciramo in v njem nastavimo naslov `pucData` na vsoto trenutnega naslova `pucBuffer` in odmika, ki je enak poravnani velikosti člena seznama (vrstici 7-8). Tako skonstruiran člen dodamo v seznam `xFreeItems` (vrstica 10).

Naredimo še korak dlje in poskusimo združiti pravkar ustvarjen blok vseh sporočil (denimo, da je njegova velikost n bajtov) s programsko strukturo `struct NamedQueue_t`. Za boljšo predstavo si pomagajmo s sliko 4.7, na kateri lahko vidimo po vrsti zložene člane te strukture znotraj enega samega bloka, pri čemer smo privzeli, da je `ucItemBuffer` polje velikosti n bajtov, v katerem bi radi hranili prej omenjen blok vseh sporočil.

Ker količine n ne moremo vnaprej izračunati (še več: pri vsaki sporočilni vrsti je lahko drugačna), je ne moremo uporabiti v definiciji strukture `struct NamedQueue_t`. V pomnilniku pa bi radi imeli točno takšno postavitev, kot je prikazana na sliki 4.7. Pomagali si bomo s konstruktom, ki je bil vpeljan s standardom C99[4] in se imenuje *prožno polje* (angl. *flexible array member*). Zasledimo ga pri definiciji člana `ucItemBuffer` (stran 76) in ga označimo z ogla-

[4]To je neformalno ime za standard ISO/IEC 9899:1999.

timi oklepaji (`[]`), torej tako kot vsako drugo polje, le da izpustimo njegovo dolžino. Prednost uporabe tega konstrukta je v dinamičnem prilagajanju velikosti zadnjega (prožnega) polja. Če rezerviramo ravno toliko pomnilnika, kolikor ga programska struktura potrebuje brez prožnega polja (označimo to velikost kot S_0), potem slednjega ne smemo uporabljati, saj mu ni preostalo nič prostora v rezerviranem bloku. Če pa rezerviramo več (denimo, da je to S_B in je $S_B > S_0$), potem lahko prožno polje uporabimo na enak način, kot če bi zanj rezervirali toliko pomnilnika, kot znaša razlika $S_B - S_0$. To najlažje ponazorimo s preprostim primerom (povzeto po standardu C99):

```
struct s { int n; double d[]; };

int m = /* poljubna vrednost */;
struct s *p = malloc(sizeof (struct s) + sizeof (double [m]));
```

Objekt, na katerega kaže `p`, lahko sedaj uporabljamo na enak način, kot če bi ga definirali z uporabo polja statično določene dolžine `m`:

```
struct { int n; double d[m]; } *p;
```

Naslednji izsek kode prikazuje rezervacijo bloka v skupnem pomnilniku za celotno sporočilno vrsto:

Izsek kode 4.6: Rezervacija celotne sporočilne vrste

```
1 UBaseType_t uxTotalSize;
2 struct NamedQueue_t *pxNamedQueue;
3 uint8_t *pucBuffer;
4
5 uxTotalSize = NQUEUE_SIZE( uxItemSize, uxQueueLength );
6 pxNamedQueue = ( struct NamedQueue_t * ) pvSharedMalloc(
7                 uxTotalSize, SHMEM_ID_NONE, 0, NULL );
8
9 pucBuffer = (*pxNamedQueue)->ucItemBuffer;
```

pri čemer smo za izračun velikosti bloka, v katerem lahko hranimo vsa sporočila skupaj s programsko strukturo `struct NamedQueue_t`, uporabili makro

NQUEUE_SIZE (izsek kode 4.4, vrstice 15-17). Za rezervacijo pomnilnika uporabimo funkcijo pvSharedMalloc iz razdelka, pri čemer bloku ne dodelimo enolične oznake, saj lahko v sistemu ustvarimo več primerkov sporočilne vrste. Izhodni parameter xNew ignoriramo in na njegovem mestu tudi podamo vrednost NULL.

Z inicializacijo sporočilne vrste nadaljujemo tako, da v vrstici 9 spremenljivki pucBuffer priredimo naslov polja ucItemBuffer (torej del bloka, kjer bomo hranili sporočila). Sedaj lahko skočimo na izsek kode 4.5, s katerim zgradimo seznam neuporabljenih sporočil.

Za konec samo še na kratko opišimo člane strukture struct NamedQueue_t, ki jih doslej še nismo omenjali. V cName, uxQueueLength in uxItemSize shranimo parametre, ki jih dobimo ob ustvarjanju vrste s funkcijo xNamedQueueCreate. Član uxReferenceCount služi kot števec referenc in ga ob ustvarjanju reference postavimo na 1, saj xNamedQueueCreate že vrne programsko ročico. Član xMutex je mutex, uxIndex pa je indeks v katalogu sporočilnih vrst. Vlogo teh članov bomo podrobneje razložili v naslednjih razdelkih.

4.2.3 Katalog sporočilnih vrst

Na začetku tega poglavja smo predstavili problem prenosa programske ročice sporočilne vrste med jedri in rešitev tega problema na nivoju programskega vmesnika. Tako smo uvedli funkcijo xQueueOpen, s katero pridobimo ročico do obstoječe sporočilne vrste, pri čemer jo moramo enolično identificirati z nekim imenom. V nadaljevanju smo tudi podrobneje opisali notranje podatkovne strukture, s katerimi hranimo sporočila in predstavimo celotno vrsto v skupnem pomnilniku. S tem znanjem lahko sedaj predstavimo mehanizem, ki se odvija v ozadju ob odpiranju in zapiranju sporočilne vrste.

Začnimo z definicijo programske ročice sporočilne vrste. V naši izvedbi je to referenca na primerek podatkovne strukture struct NamedQueue_t v skupnem pomnilniku in jo imenujemo NamedQueueHandle_t.

Operacija odpiranja od uporabnika zahteva, da kot parameter poda enolično ime vrste. Nato moramo v skupnem pomnilniku poiskati primerek

sporočilne vrste, katerega ime (član `pcName`) se ujema s tem imenom. Če takšen primerek obstaja, potem kot rezultat vrnemo njegovo referenco, še prej pa mu povečamo števec referenc `uxReferenceCount`.

Iskanje sporočil v skupnem pomnilniku omogočimo z uporabo kataloga, ki ga prav tako hranimo v skupnem pomnilniku in ga definiramo kot polje programskih ročic `pxNamedQueues` znotraj naslednje programske strukture:

Izsek kode 4.7: Programska struktura za katalog sporočil

```
struct NamedQueueIndex_t
{
    portMUTEX_TYPE xMutex;
    struct NamedQueue_t *pxNamedQueues[];
};
```

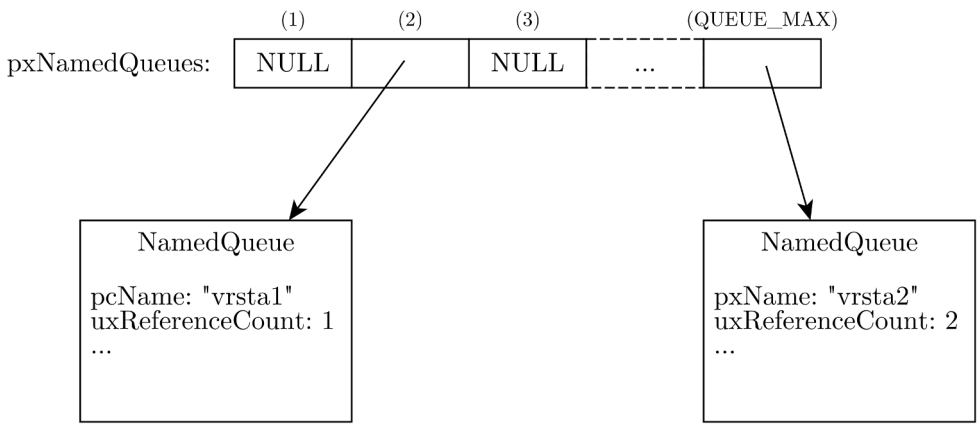
Vsakič, ko sporočilno vrsto ustvarimo, shranimo njeno programsko ročico v enega izmed prostih elementov polja `pxNamedQueues`, pri čemer prost element naznanimo z vrednostjo `NULL`. Na sliki 4.8 imamo primer kataloga, v katerem sta dodani dve sporočilni vrsti. Eno smo poimenovali "vrsta1"in je bila pravkar ustvarjena, pri čemer se je njena programska ročica shranila v element kataloga z zaporedno številko 2. Druga vrsta z imenom "vrsta2"je bila še dodatno odprta iz nekega drugega opravila. V katalogu se je njena programska ročica shranila v zadnji element (zaporedna številka `QUEUE_MAX`).

Rešitev s katalogom sporočil odpira par vprašanj, ki jih moramo nasloviti. Najprej moramo ugotoviti, kje se v skupnem pomnilniku nahaja katalog. Tu nam priskoči na pomoč označevanje blokov v skupnem pomnilniku, ki smo ga predstavili v razdelku 4.2.1.

Ko rezerviramo pomnilniški blok kataloga, mu dodelimo neko enolično oznako (recimo ji `QUEUE_BLOCK_ID`), kot je to prikazano z naslednjim izsekom kode:

Izsek kode 4.8: Rezervacija pomnilniškega bloka za katalog sporočilnih vrst

```
pxNamedQueueIndex = pvSharedMalloc( NQUEUE_INDEX_SIZE( QUEUE_MAX ),
                                    QUEUE_BLOCK_ID, 0, &xNew );
```



Slika 4.8: Primer kataloga sporočilnih vrst

Ker je sedaj pomnilniški blok s katalogom enolično označen, lahko referenco nanj dobimo ob vsakem naknadnem klicu funkcije `pvSharedMalloc`. Zaradi boljšega razumevanja izseka kode 4.8 dodajmo še naslednje definicije:

Izsek kode 4.9: Dodatni parametri kataloga sporočil

```
1  #ifndef QUEUE_MAX
2  #  define QUEUE_MAX     ( ( size_t ) 64 )
3  #endif
4
5  #define QUEUE_BLOCK_ID  ( ( uint32_t ) 0x6e71 )
6
7  #define NQUEUE_INDEX_SIZE( n ) (                \
8      sizeof( struct NamedQueueIndex_t ) +        \
9      ( sizeof( struct NamedQueueIndex_t * ) * n ) \
10 )
```

Naša programska izvedba skupnega pomnilnika sama po sebi ne zagotavlja enoličnosti oznak, zato moramo za njihovo dodeljevanje znotraj sistema poskrbeti sami. Ker je sporočilna vrsta trenutno edini uporabnik skupnega pomnilnika, si vzemimo nekaj več svobode in izberimo enolično oznako 0x6e71 (izsek kode 4.9, vrstica 5), kar je po ASCII kodi bajtni zapis za znaka

'n' in 'q' (kot kratica za „named queue“).

Velikost kataloga je fiksna in je omejena z maksimalnim številom sporočilnih vrst, ki jih imamo lahko naenkrat odprtih. To količino definiramo v vrsticah 1-3 kot `QUEUE_MAX` in njeno privzeto vrednost nastavimo na 64.

Princip rezervacije pomnilniškega bloka za katalog je podoben tistemu, ki smo ga predstavili pri sporočilnih vrstah, zato bomo njegovo podrobno razlago izpustili.

Dodatna težava, ki jo moramo rešiti, je primer, ko več opravil sočasno ustvarja, odpira in zapira sporočilne vrste. To pomeni, da bodo opravila sočasno dostopala do kataloga v skupnem pomnilniku in ga moramo zato ustrezno zaščititi. V ta namen uporabimo mutex, ki smo ga zamolčali ob predstavitvi strukture `NamedQueueIndex_t` v izseku kode 4.7.

4.2.4 Pošiljanje in sprejemanje sporočil

Najprej bomo s psevdokodo pokazali, kako z uporabo doslej predstavljenih programskih funkcij in struktur poteka pošiljanje in sprejemanje sporočil. Nato bomo ta opis nadgradili z nekaj podrobnostmi iz programske kode.

Dodajanje sporočila M prioritete p v sporočilno vrsto Q poteka na način, kot ga prikazujemo s funkcijo **MsgSendToBack**. V vrstici 2 odstranimo prvi člen iz seznama neuporabljenih sporočil. V dobljeni člen nato prekopiramo podatke izvornega sporočila M in ga nazadnje dodamo na konec seznama uporabljenih sporočil s prioriteto p (vrstici 4-5). Pri tem smo predpostavili, da imamo na voljo operacijo $memcpy(B, A, n)$, ki kopira vsebino velikosti n bajtov iz pomnilniške lokacije A v B. V primeru, ko želimo dodati sporočilo po načinu LIFO, je postopek enak, le da namesto funkcije *itemListAppend* uporabimo funkcijo *itemListPrepend* za dodajanje na začetek seznama.

Ob jemanju sporočila z vnaprej znano prioriteto p storimo ravno obratno. V vrstici 2 funkcije **MsgReceiveWithPriority** odstranimo prvi člen iz seznama uporabljenih sporočil s prioriteto p, nato pa v vrsticah 4-5 prekopiramo vsebino sporočila iz odvzetega člena v medpomnilnik sporočila M in prosti člen dodamo v seznam neuporabljenih sporočil. Tu je v principu

Algoritem MsgSendToBack (pošlji sporočilo na konec vrste)

```
1: function MsgSendToBack(Q, M, p)
2:     item ← itemListRemove(Q.xFreeItems)
3:     if item ≠ NULL then
4:         memcpy(item.pucData, M, Q.uxItemSize)
5:         itemListAppend(Q.xUsedItems[p], item)
6:         return TRUE
7:     else
8:         return FALSE
```

vseeno, ali dodajamo na konec ali začetek seznama. Ker vedno odvzemamo člene iz začetka seznama, smo se odločili za drugo možnost, kajti verjamemo, da bomo z njo utegnili doseči nekoliko večjo stopnjo lokalnosti pomnilniških dostopov. Tako bomo ob naslednjem jemanju člena iz seznama vedno vzeli tistega, ki smo ga vanj nazadnje dodali.

Na podoben način kot sprejemanje realiziramo operacijo „pokukaj“ (funkcija **MsgPeekWithPriority**), le da člena iz seznama uporabljenih sporočil ne prenesemo v seznam neuporabljenih sporočil, temveč zgolj pridobimo referenco nanj (vrstica 2).

Algoritem MsgReceiveWithPriority (sprejmi sporočilo z določeno prioriteto)

```
1: function MsgReceiveWithPriority(Q, M, p)
2:     item ← itemListRemove(Q.xUsedItems[p])
3:     if item ≠ NULL then
4:         memcpy(M, item.pucData, Q.uxItemSize)
5:         itemListPrepend(Q.xFreeItems, item)
6:         return M
7:     else
8:         return NULL
```

Prikažimo še postopek sprejemanja, ko prioritete ne podamo eksplicitno, temveč hočemo sprejeti prvo sporočilo z najvišjo prioriteto. Osnovna ideja

Algoritem MsgPeekWithPriority (pokukaj v sporočilo z določeno prioriteto)

```
1: function MsgPeekWithPriority(Q, M, p)
2:     item ← itemListFirst(Q.xUsedItems[p])
3:     if item ≠ NULL then
4:         memcpy(M, item.pucData, Q.uxItemSize)
5:         return M
6:     else
7:         return NULL
```

je, da v padajočem vrstnem redu prioritete linearno iščemo po seznamih uporabljenih sporočil, dokler ne najdemo prvega nepraznega. Od tu naprej je postopek enak kot pri **MsgReceiveWithPriority**, kar zapišemo z naslednjo psevdokodo:

Algoritem MsgReceive (sprejmi sporočilo)

```
1: function MsgReceive(Q, M)
2:     for p ← ItemPriorityCount − 1, 0 do
3:         item ← itemListRemove(Q.xUsedItems[p])
4:         if item ≠ NULL then
5:             memcpy(M, item.pucData, Q.uxItemSize)
6:             itemListPrepend(Q.xFreeItems, item)
7:             return M
8:     return NULL
```

Podobno naredimo pri operaciji pokukaj (funkcija **MsgPeek**), le da namesto *itemListRemove* uporabimo *itemListFirst*. Ker gre za praktično enako spremembo kot prej pri **MsgReceiveWithPriority**, bomo njeno psevdokodo izpustili.

Na tej točki moramo priznati, da z linearnim iskanjem nekoliko povečamo časovno zahtevnost operacije sprejemanja sporočil. V praksi pa se to ne izkaže kot velik problem, saj je število prioritet fiksno in vnaprej znano, zato je možno vnaprej predvideti čas izvajanja za najbolj neugoden primer, ko

opravimo sprehod po celem seznamu. Poleg tega imamo opravka z relativno majhnim številom prioritet (v našem primeru 5), zato ta čas tudi v najbolj neugodnem primeru ne bo zelo dolg. Iskanju pa se povsem izognemo, če eksplicitno podamo prioriteto sporočila, ki ga želimo sprejeti.

Izvedba monitorja

Izsek kode 7.3 (Dodatek, str. 108) prikazuje funkcijo pošiljanja sporočila `prvSendPrio`. V osnovi gre za dodajanje sporočila v vrsto, kot smo ga že definirali s funkcijo **MsgSendToBack**, kar lahko vidimo v vrsticah 24-31, kjer se glede na parameter `xSendToBack` le še odločamo, ali bomo pošiljali na konec ali na začetek vrste.

Kot smo že razložili v razdelku 2.2.4, lahko na pošiljanje sporočila gledamo kot na operacijo P semaforja, pri čemer vrednosti semaforja sedaj ustreza število članov v seznamu neuporabljenih sporočil `xFreeItems` (brez stražarjev). Kot smo to že izpostavili pri obravnavi semaforjev, se mora operacija P izvesti atomarno in s to idejo tudi naredimo kritično območje med vrsticami 16 in 48, ki ga ščitimo z mutex-om. Vendar pa funkcija dodajanja sporočila, ki jo kličemo v vrstici 26 oziroma 30, za razliko od operacije P ne blokira izvajanja.

Da bi lažje razumeli, zakaj smo se odločili za nekoliko ,,pomanjkljivo“ izvedbo funkcije dodajanja, si najprej oglejmo, kakšen bi bil potek, ko bi ta funkcija blokirala. Ker jo kličemo znotraj kritičnega območja, pomeni, da bo v njem čakala vse dokler se v sporočilni vrsti ne pojavi prosto mesto za novo sporočilo. A to se ne bo zgodilo, kajti prosto mesto moramo šele ustvariti iz nekega drugega opravila s klicem funkcije za sprejemanje, ki pa bo obtičala na vstopu v že zaklenjeno kritično območje. Posledično torej pride do tako imenovanega ,,smrtnega objema“, kjer opravila v nedogled čakajo drugo drugega.

Situacija je natanko takšna, kot smo jo opisali pri obravnavi monitorjev v razdelku 2.2.6. Zato bomo uporabili predlagano rešitev iz istega razdelka in bomo namesto blokiranja začasno izstopili iz kritičnega območja ter opravilo

spravili v stanje čakanja. To bomo storili z uporabo naslednje funkcije:

```
void vTaskPlaceOnEventList( xList *pxEventList,
                            portTickType xTicksToWait );
```

Ta funkcija je del sistema FreeRTOS in deluje na naslednji način: imamo seznam `pxEventList`, v katerega dodajamo opravila za čakanje. Vsem tem opravilom je skupno, da čakajo na isto vrsto dogodka. Opravilo, ki se želi postaviti v čakanje, mora sámo poklicati funkcijo `vTaskPlaceOnEventList`. Ta ga bo uvrstila na ustrezno mesto v seznamu `pxEventList`, ki je urejen po prioritetah izvajanja opravil. Funkcija bo tudi poskrbela, da bo opravilo ob naslednjem preklopu razporejevalnika prešlo v eno izmed naslednjih dveh stanj: *suspendirano* (če je parameter `xTicksToWait` enak `portMAX_DELAY`) ali *čakanje* (v vseh drugih primerih). Na ta način smo implementirali opreracijo *wait* monitorja.

Sedaj še poglejmo, kako je realizirana operacija *signal*. Opravilo, ki na pravkar opisani način preide v stanje *suspendirano* ali *čakanje*, spravimo v stanje *pripravljeno* tako, da iz nekega drugega opravila kličemo funkcijo `vTaskRemoveFromEventList`:

```
BaseType_t xTaskRemoveFromEventList( xList *pxEventList );
```

S to funkcijo pošljemo signal enemu čakajočemu opravilu, da se je zgodil dogodek, na katerega čaka. Katero izmed čakajočih opravil bo izbrano, je odvisno od razporeditve znotraj seznama `pxEventList`, za katerega smo že rekli, da je urejen po prioriteti opravil. Funkcija tako vzame prvo opravilo iz seznama `pxEventList` in ga postavi v stanje *pripravljeno*. Posledično se bo opravilo pričelo izvajati v enem izmed naslednjih preklopov razporejevalnika. Kdaj točno, je seveda odvisno od načina delovanja razporejevalnika in preostalih opravil, ki se razvrščajo.

Opravilo, ki je v stanju *čakanje*, lahko preide nazaj v stanje *pripravljeno* tudi ob odsotnosti opazovanega dogodka. Če ga v času `xTicksToWait` urinih period ne bo nihče drug signaliziral s funkcijo `vTaskRemoveFromEventList`, potem se bo avtomatično pričelo izvajati po izteku tega časa.

Podrobna razlaga izvorne kode

Da bi še bolje razumeli programsko izvedbo pošiljanja in sprejemanja sporočil, moramo natančneje opredeliti dogodke in s tem tudi število primerkov seznamov, ki jih bomo uporabili za dodajanje in jemanje opravil. Spomnimo, da v posamičen seznam dodajamo le opravila, ki čakajo na isto vrsto dogodka. Dogodek, na katerega bo čakala funkcija `prvSendPrio`, je sproščeni prostor v sporočilni vrsti. Seznam, v katerega dodajamo opravila, ki čakajo na to vrsto dogodka, smo poimenovali `xTaskSendList`. Funkcija `prvReceive` (izsek kode 7.4) je na nek način zrcalna slika `prvSendPrio`, zato je doslej nismo posebej omenjali. S stališča dogodkov pa vseeno povejmo, da bo čakala na dogodek, ko se v prazni vrsti pojavi novo sporočilo. Ime seznama opravil za to vrsto dogodkov je `xTaskRecvList`.

Sedaj moramo še premisliti, ali za celotni sistem zadošča en sam par seznamov `xTaskSendList` in `xTaskRecvList`. Denimo, da se dve opravili T_a in T_b dodata na isti seznam `xTaskSendList`, pri čemer opravilo T_a čaka na sporočilni vrsti Q_a, opravilo T_b pa na Q_b in je vrstni red v seznamu (T_a, T_b). Sedaj se v nekem tretjem opravilu izvede funkcija `prvReceive` na vrsti Q_b, ki bo prvemu opravilu iz seznama `xTaskSendList` signalizirala, da se je v vrsti pojavilo prosto mesto. Ker je prvo opravilo T_a, bo to ponovno poskusilo dodati sporočilo v vrsto Q_a, a bo pri tem neuspešno, saj se je eno mesto sprostilo v vrsti Q_b in ne v Q_a. Tako se bo T_a ponovno dodalo na seznam čakanja, medtem ko bo T_b zgrešilo dogodek in se zato ne bo izvedlo. Očitno rabimo po en par za vsako sporočilno vrsto posebej, zato ustvarimo polji seznamov `xTaskSendList` in `xTaskRecvList` dolžine `QUEUE_MAX`. Indeksi elementov v teh dveh poljih se ujemajo z indeksi sporočilnih vrst v katalogu sporočil. Drugače povedano, i-ti sporočilni vrsti v katalogu ustrezata i-ta seznama v poljih `xTaskSendList` in `xTaskRecvList`.

Polji seznamov `xTaskSendList` in `xTaskRecvList` ne shranjujemo v skupni pomnilnik, kot bi sprva morda predpostavili. Poglejmo, zakaj. Vsak tak seznam vsebuje opravila, ki so definirana znotraj enega primerka operacijskega sistema. Ker na našem sistemu tečeta dva primerka sistema FreeRTOS

(na vsakem procesorskem jedru en primerek), se bodo opravila iz različnih operacijskih sistemov pričela mešati znotraj istega seznama. Tako se bodo na vsakem sistemu pričela razporejati tudi „tuja“ opravila in je delovanje sistema v takšnem primeru nedefinirano. Zato moramo te sezname ločiti in na vsakem jedru uporabljati lokalne primerke.

Na podlagi doslej povedanega lahko dokončno razložimo pomemben del funkcije `prvSendPrio` (in posledično tudi `prvReceive`). Če dodajanje sporočila v vrsto zaradi nekega razloga ne uspe, bo spremenljivka `xResult` enaka `pdFALSE` in se v tem primeru v vrstici 51 vprašamo, če dovolimo čakanje. Če ne, potem je parameter `xTicksToWait` enak `0`, zato preskočimo vrstice 52-71 in ker časovnika sploh še nismo pognali (spremenljivka `xTimerRunning`, ki smo jo v vrstici 11 postavili na `pdFALSE`), takoj izstopimo iz zanke in tudi iz celotne funkcije (vrstice 73-75).

Če funkcija sme čakati, potem bomo najprej shranili trenutno časovno značko in s tem aktivirali časovnik, razen če tega nismo storili že prej (vrstice 53-57). Nato izmerimo čas, ki je pretekel od trenutka aktivacije časovnika in ga primerjamo s podano omejitvijo `xTicksToWait` (vrstica 59). Če je preteklo manj kot `xTicksToWait` časa, opravilo dodamo na seznam čakanja z že znano funkcijo `vTaskPlaceOnEventList` (vrstici 62-63). Ker funkcije nad istim seznamom ne smemo uporabljati iz več opravil hkrati, jo ovijemo v kritično območje, ki ga tokrat zaščitimo zgolj z onemogočanjem prekinitev (vrstici 61 in 65). To zadošča, kajti seznam ni v skupnem pomnilniku, temveč je lokalen znotraj jedra. V vrstici 64 pokličemo še funkcijo `portYIELD_WITHIN_API`, zaradi katere se bo sprožil preklop razporejevalnika takoj ob izstopu iz kritičnega območja v vrstici 65. Ko se bo opravilo ponovno pričelo izvajati, se bo le še preizkusil pogoj v vrstici 73 in se bomo zaradi aktiviranega časovnika (`xTimerRunning` je enak `pdTRUE`) ponovno vrnili na začetek zanke.

Poglejmo še, kaj se zgodi ob uspešnem dodajanju sporočila v vrsto. V tem primeru bo pogoj v vrstici 34 izpolnjen in s funkcijo `xTaskRemoveFromEventList` bomo signalizirali prvemu opravilu, ki na tej sporočilni vrsti čaka v funkciji za sprejemanje sporočila. Če ima to opravilo višjo prioriteto od našega, bo

funkcija vrnila vrednost `pdTRUE` in zato podobno kot v vrstici 64 pokličemo funkcijo `portYIELD_WITHIN_API`, ki bo povzročila preklop razporejevalnika takoj ob izstopu iz kritičnega območja v vrstici 48. Dodajmo še, da funkcija `xTaskRemoveFromEventList` zahteva, da pred uporabo eksplicitno preverimo, če seznam ni prazen, od tod tudi pogojni stavek v vrstici 36.

Signalizacija med jedri

Ker sta seznama `xTaskSendList` in `xTaskRecvList` lokalna znotraj vsakega jedra, moramo rešiti še en problem - kako ob nekem dogodku signalizirati čakajočim opravilom iz seznama na nasprotnem jedru. Podobno kot proizvajalec mikrokrmilnika predlaga v svojem protokolu, si bomo pomagali s prekinitvami. Ko v funkciji `prvSendPrio` signaliziramo dogodek, bomo v vrstici 45 sprožili še prekinitev s klicem funkcije `prvSendEvent` (celotno funkcijo `prvSendEvent` prikazuje izsek kode 7.5). Pred samim proženjem prekinitve pa v posebej rezerviran prostor v skupnem pomnilniku shranimo informacije o dogodku, ki jo je povzročil. To informacijo predstavimo z naslednjo programsko strukturo:

```
1 struct NamedQueueEvent_t
2 {
3     UBaseType_t uxIndex;
4     enum EventType_t xEventType;
5     BaseType_t xPending;
6 };
```

V `uxIndex` shranimo indeks sporočilne vrste v katalogu, znotraj katere se je sprožil dogodek. Vrsto dogodka (pošiljanje ali sprejemanje) shranimo v `xEventType`, `xPending` pa je zastavica, ki jo dvignemo ob proženju prekinitve (vrednost `pdTRUE`) in tako označimo, da dogodek še ni bil servisiran v nasprotnem jedru. Ko prekinitveno servisna rutina (prikazana v izseku kode 7.6) v nasprotnem jedru konča s servisiranjem, spusti zastavico (vrednost `pdFALSE`) in tako omogoči vnovično signaliziranje. Način, kako zastavica prepreči nadaljnje dodajanje sporočil v vrsto, lahko vidimo v vrstici 20 funkcije

`prvSendPrio`.

Naloga prekinitveno servisne rutine je poleg spuščanja zastavice tudi ustrezno signaliziranje opravil znotraj jedra. Glede na vrsto dogodka (pošiljanje ali sprejemanje) izberemo ustrezen seznam opravil, in sicer:

- če je dogodek vrste `SendEvent`, potem ga je oddala funkcija `prvSendPrio` in zato aktiviramo prvo med opravili, ki čakajo v funkciji za sprejemanje (izsek kode 7.6, vrstice 12-15),
- če je dogodek vrste `ReceiveEvent`, potem ga je oddala funkcija `prvReceive` in aktiviramo prvo med opravili, ki čakajo v funkciji za sprejemanje (vrstice 19-22).

Pripomnimo, da dokumentacija v izvorni kodi sistema FreeRTOS navaja, da lahko funkcijo `xTaskRemoveFromEventList` kličemo znotraj prekinitveno servisne rutine.

4.3 Izvedba drugih programskih modulov

4.3.1 Skupni pomnilnik

Funkcija `pvSharedMalloc` operira nad pomnilniškim blokom, ki ga definiramo z začetnim naslovom in dolžino v bajtih. V ta namen vpeljemo simbola `configSHMEM_BASE` in `configSHMEM_SIZE`, ki ju definiramo v času prevajanja.

Spomnimo, da funkcijo `pvSharedMalloc` kličemo tako pri ustvarjanju kot pri odpiranju sporočilne vrste. Zato se lahko zgodi, da bo klicana sočasno iz več opravil in posledično moramo poskrbeti, da se rezervacija pomnilnika in tudi iskanje pomnilniškega bloka s podano oznako izvedeta atomarno. V ta namen uvedemo mutex, ki ga eksplicitno inicializiramo ob zagonu sistema s funkcijo `pvSharedInit`.

Proste in zasedene bloke v deljenem pomnilniku povežemo v svoj seznam prostih oziroma zasedenih blokov. Bloke rezerviramo z uporabo algoritma *najboljše prileganje* (angl. *best fit*), med sproščanjem pa jih povezujemo z morebitnimi sosednjimi prostimi bloki.

4.3.2 Mutex

Mutex smo implementirali s Petersonovim algoritmom. Razlog za takšno odločitev leži v pomožnem jedru, ki je vrste ARM Cortex-M0. Slednje ne vsebuje strojnih konstruktov, ki bi jih lahko uporabili v ta namen.

Algoritem, ki smo ga spoznali v razdelku 2.2.5, lahko uporabimo za medsebojno izključitev zgolj dveh opravil. Na našem sistemu pa imamo dvoje jeder, pri čemer se lahko na vsakem izvaja več opravil. V naši implementaciji mutex-a zato pred samim začetkom vstopnega protokola (**PetersonLock**) najprej onemogočimo prekinitve, s čimer na posameznem jedru reduciramo izvajanje na zgolj eno opravilo. Prekinitve znova omogočimo ob koncu izstopnega protokola.

Poglavje 5

Prikaz uporabe

5.1 Opis aplikacije

Kot dokaz koncepta delovanja naše rešitve smo izdelali napravo, ki omogoča vklop/izklop zunanje enote glede na izmerjeno temperaturo okolice. Naprava je sestavljena iz:

- temperaturnega senzorja LM35 za merjenje temperature okolice,
- releja za vklop/izklop zunanje enote,
- razvojne plošče Hitex LPC4350 z mikrokrmilnikom LPC4350.

Krmiljenje poteka tako, da naprava periodično vzorči signal iz temperaturnega senzorja, vrednost vzorca pretvori v temperaturo, izraženo v stopinjah Celzija ter dobljeno temperaturo primerja s prejšnjo vrednostjo ter mejnima vrednostma, ki ju lahko med delovanjem poljubno spreminjamo. Če temperatura naraste nad zgornjo mejno vrednost, preklopimo rele. Podobno preklopimo rele tudi v primeru, ko temperatura pade pod spodnjo mejno vrednost. Z uvedbo spodnje in zgornje mejne vrednosti preprečimo hitre preklope releja, ki bi se sicer dogajali, če bi odčitki temperature nihali okrog ene same mejne vrednosti.

Nastavljanje naprave poteka preko terminala na vmesniku UART. Nastavljamo lahko obe mejni vrednosti v stopinjah Celzija, periodo vzorčenja

v sekundah, ter pozitivno/negativno logiko releja. Če nastavimo pozitivno logiko, je rele vklopljen, ko je presežena zgornja mejna vrednost.

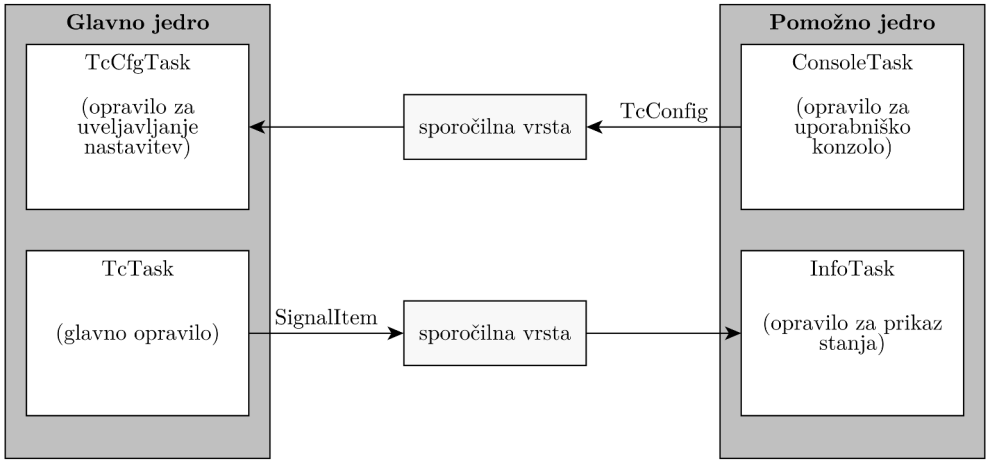
5.2 Opis izvedbe

Pripravili smo ločen primerek operacijskega sistema FreeRTOS za vsako jedro posebej. Osnovna ideja je, da na glavnem jedru teče časovno kritična koda, medtem ko se bo na pomožnem jedru izvajala koda uporabniškega vmesnika in prikaza stanja z LED diodami.

Najprej opišimo delovanje časovno kritične kode. Ta mora ob strogo določeni periodi vzorčiti signal iz analogno-digitalnega pretvornika (ADC), na katerega je priključen temperaturni senzor, nato pa preračunati to temperaturo v stopinje Celzija in jo tudi povprečiti, s čimer bomo zmanjšali šum in posledično tudi naključne oscilacije, ki se zaradi zunanjih motenj pojavljajo v analognem signalu. Dobljeno povprečno vrednost mora nato primerjati z mejnimi temperaturami, ki sodijo v sistemske nastavitve in jih uporabnik lahko spremeni kadarkoli med delovanjem sistema. Glede na rezultat te primerjave in nastavitev pozitivne/negativne logike ustrezno krmili rele.

Na pomožnem jedru realiziramo uporabniški vmesnik, ki preko vmesnika UART izpisuje trenutno temperaturo v stopinjah Celzija. Uporabnik lahko s pritiskom tipke ,,Enter“ kadarkoli sproži urejanje nastavitev temperaturnega krmilnika. Takrat se trenutna temperatura preneha izpisovati, saj se namesto nje prikaže urejevalnik nastavitev, Ko uporabnik potrdi nove nastavitve, jih dostavimo glavnemu jedru. Pomožno jedro dodatno še poskrbi, da je stanje releja vedno vidno s pomočjo LED diod (zelena: vklopljen, rdeča: izklopljen). Stanje LED diod se osvežuje tudi med urejanjem nastavitev.

V nadaljevanju sledi podrobnejši opis vseh sistemskih opravil, ki jih glede na njihov položaj v dvoprocesorskem sistemu prikazujemo s sliko 5.1.



Slika 5.1: Sistemska opravila temperaturnega krmilnika

Opravilo za uporabniško konzolo

To opravilo se prične izvajati takoj ob zagonu pomožnega jedra. Opravilo bo najprej nek kratek čas pred zagonom temperaturnega krmilnika (približno 3 sekunde) ponudilo uporabniku možnost spremembe nastavitev (izsek kode 7.9, vrstica 12). Če uporabnik v tem času pritisne tipko ,,Enter“, se odpre urejevalnik nastavitev (vrstica 16), sicer se izpišejo privzete nastavitve (vrstice 20-22). V vrstici 25 se nastavitve temperaturnega krmilnika z uporabo naše sporočilne vrste prvič pošljejo glavnemu jedru, s čimer se, kot bomo kasneje videli, zažene temperaturni krmilnik. V nadaljevanju se v zanki bere vhod iz vmesnika UART (vrstica 29) in ob pritisku na tipko ,,Enter“ se znova odpre urejevalnik nastavitev (vrstica 32). Potrjene nove nastavitve se nato pošljejo glavnemu jedru (vrstica 33).

Sporočilo z nastavitvami, ki se pošljejo glavnemu jedru, definiramo z naslednjo strukturo:

```
struct TcConfig_t
{
    uint16_t   xTempLo;
    uint16_t   xTempHi;
```

```
    BaseType_t xIsPositive;
};
```

pri čemer spodnji temperaturni prag shranimo v `xTempLo`, zgornji prag v `xTempHi`, nastavitev pozitivne/negativne logike pa v `xIsPositive`.

Opravilo za uveljavljanje nastavitev

To opravilo se izvaja na glavnem jedru in je nižje prioritete kot opravilo, v katerem teče kritična koda. Razlog, da smo to opravilo sploh ustvarili, je v prejemu nastavitev iz pomožnega jedra na tak način, ki ne bo vplival na izvajanje časovno kritične kode. Ob zagonu sistema je to opravilo edino, ki se izvaja na glavnem jedru. Ko prvič prejme nastavitve iz pomožnega jedra, jih shrani v globalno spremenljivko `xTcConfig` in nato spravi glavno opravilo v stanje izvajanja (izsek kode 7.8, vrstica 18). Ob vsakem naslednjem prejemu nastavitev jih to opravilo le še shranjuje v globalno spremenljivko.

Glavno opravilo

Glavno opravilo ob vsakem obhodu zanke najprej pobere morebitne nove nastavitve iz globalne spremenljivke `xTcConfig` (izsek kode 7.7, vrstice 15-17). V vrsticah 19-20 nato pridobi nov vzorec iz ADC pretvornika in izračuna povprečje zadnjih 32 vzorcev. Če je bil sistem na novo zagnan in je zgodovini manj kot 32 vzorcev, bo funkcija `prvGetNewAverage` vrnila vrednost `UINT16_MAX`. V tem primeru opravilo preide v stanje čakanja (vrstica 48) in tam prebije toliko časa, kot znaša razlika med njegovo periodo (nastavili smo jo na 200 ms) in časom, ko se je zanka nazadnje pričela izvajati. Tako dosežemo večjo natančnost periode.

V primeru, ko je povprečna vrednost veljavna, jo pretvorimo v stopinje Celzija (vrstica 24) in nato določimo stanje releja glede na presežen zgornji/-spodnji prag in pozitivno/negativno logiko (vrstice 26-33).

Sedaj moramo še poslati zadnjo izračunano povprečno temperaturo in zadnje stanje releja pomožnemu jedru, ki bo te informacije prikazalo na

uporabniškem vmesniku (vrstica 37). Pri tem uporabljamo našo izvedbo sporočilne vrste, za samo sporočilo pa definiramo naslednjo strukturo:

```
struct SignalItem_t
{
    uint16_t  xTemperature;
    BaseType_t xPowerOn;
};
```

pri čemer je `xTemperature` zadnja povprečna temperatura v stopinjah Celzija, `xPowerOn` pa stanje releja (vrednost `pdTRUE`: vklopljen, vrednost `pdFALSE`: izklopljen).

Da uporabniškega vmesnika ne preobremenimo po nepotrebnem, sporočila pošiljamo z večkratnikom periode vzorčenja (`TC_REFRESH_PERIOD` in spremenljivka `xRefreshCounter`).

Na koncu še krmilimo rele glede na prej izračunano vrednost `xPowerOn` (vrstica 45).

Opravilo za prikaz stanja

To opravilo teče na pomožnem jedru. Njegov namen je, da uporabnika obvešča o trenutnem stanju temperaturnega krmilnika. Opravilo prejema sporočila tipa `struct SignalItem_t` iz glavnega opravila. Če uporabnik nima odprtega urejevalnika nastavitev, se zadnje prejeto sporočilo ustrezno oblikuje in izpiše na konzolo (vrstice 48-54). Ne glede na stanje urejevalnika pa se posodabljajo LED indikatorji (vrstice 45-46 in 56-59).

Poglavje 6

Zaključek

V tem diplomskem delu smo glede na zastavljeno tematiko naloge najprej preučili motive za obstoj asimetričnih večjedrnih mikrokrmilnikov. Na ta način smo želeli bolje razumeti pomen in uporabno vrednost našega dela. Spoznali smo, da je bila dodatna spodbuda za razvoj večjedrnih sistemov energetska učinkovitost ob visoki stopnji vzporednosti, kar je pripeljalo do njihove širše uporabe. Zato jih lahko pogosto zasledimo v vgrajenih sistemih, kjer so lahko koristni ne le zaradi svoje energetske učinkovitosti, temveč tudi zaradi možnosti sočasnega izvajanja opravil in celo sočasnega izvajanja različnih primerkov operacijskih sistemov. Tako lahko izoliramo izvajanje časovno kritičnih funkcij na enem jedru, medtem ko pomožno jedro uporabljamo za različne podporne funkcije.

Cilj tega diplomskega dela je bil implementirati mehanizem komunikacije v asimetričnem dvojedrnem sistemu LPC4350. Ker gre za sistem z razmeroma omejenimi sistemskimi viri, smo kot operacijski sistem izbrali FreeRTOS. Ta operacijski sistem je pogosto uporabljen v vgrajenih sistemih z omejenimi viri, obenem pa omogoča razvoj rešitev, ki delujejo v stvarnem času. Kot mehanizem za komunikacijo smo implementirali sporočilno vrsto, ki jo sedaj lahko uporabimo za usklajevanje in izmenjavo informacij med opravili, ki tečejo na različnih jedrih in posledično tudi na različnih primerkih sistema FreeRTOS. V tem smislu gre za nadgradnjo obstoječih sporočilnih

vrst na sistemu FreeRTOS. Sporočilno vrsto smo implementirali tako, da omogoča pošiljanje in sprejemanje sporočil na časovno predvidljiv način in v tem smislu je enakovredna izvirni sporočilni vrsti na sistemu FreeRTOS.

Kot primer uporabe smo na isti strojni platformi, to je LPC4350, razvili temperaturni krmilnik, ki na podlagi trenutno izmerjene temperature okolice krmili zunanji rele. Temperaturni krmilnik smo razvili tako, da na pomožnem jedru teče uporabniški vmesnik, medtem ko se na glavnem jedru izvaja časovno kritična funkcija enakomernega vzorčenja temperature in hitrega preklopa releja ob preseženih mejnih vrednostih. Za komunikacijo med programskimi opravili na glavnem in pomožnem jedru smo uporabili našo izvedbo sporočilnih vrst, pri čemer smo uspeli doseči, da z izmenjavo sporočil ne oviramo časovno kritične funkcije v izvajanju ob natančno določeni periodi.

V nadaljevanju bi lahko našo rešitev uporabili na nekem resničnem časovno kritičnem sistemu v industriji. V smislu nadaljnjega razvoja rešitve bi bilo smiselno preučiti možnost izvedbe na sistemih z več kot dvema jedroma.

Poglavje 7

Dodatek

7.1 Sporočilne vrste na sistemu FreeRTOS

Izsek kode 7.1: Primer sočasne uporabe sporočilne vrste iz štirih opravil

```
1  struct xTaskMessage
2  {
3      char cReceiver[ configMAX_TASK_NAME_LEN ];
4      char cSender[ configMAX_TASK_NAME_LEN ];
5      long lValue;
6  };
7
8  struct xProducerContext
9  {
10     char cReceiver[ configMAX_TASK_NAME_LEN ];
11     xQueueHandle xQueue;
12 };
13
14 static void vProducer( void *pvParameters )
15 {
16     struct xProducerContext *xContext;
17     struct xTaskMessage xMessage;
18     portBASE_TYPE xStatus;
19     long lIterCount;
20
```

```
21      xContext = ( struct xProducerContext * ) pvParameters;
22
23      strcpy( xMessage.cReceiver, xContext->cReceiver );
24      strcpy( xMessage.cSender, pcTaskGetTaskName( NULL ) );
25
26      lIterCount = 0;
27
28      for( ; ; )
29      {
30          xMessage.lValue = lIterCount;
31          xStatus = xQueueSendToBack( xContext->xQueue,
32              ( void * ) &xMessage, 0 );
33          if( xStatus != pdPASS )
34          {
35              UARTPrintf( "Posiljanje sporocila ni uspelo.\r\n" );
36          }
37
38          ++lIterCount;
39      }
40  }
41
42  static void vConsumer( void *pvParameters )
43  {
44      xQueueHandle xQueue;
45      struct xTaskMessage xMessage;
46      portBASE_TYPE xStatus;
47      const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
48
49      xQueue = ( xQueueHandle ) pvParameters;
50
51      for( ; ; )
52      {
53          /* Preveri, ce vrsta ni prazna in je naslednje sporocilo namenjeno
54             temu opravilu. */
55
56          xStatus = xQueuePeek( xQueue, ( void * ) &xMessage, xTicksToWait );
57          if( xStatus == pdPASS &&
58              strcmp( xMessage.cSender, pcTaskGetTaskName( NULL ) ) == 0 )
```

```
59          {
60              /* Naslovnik je pravi, poberi sporocilo iz vrste. */
61
62              xStatus = xQueueReceive( xQueue, ( void * ) &xMessage, 0 );
63              if( xStatus == pdPASS )
64              {
65                  UARTPrintf( "Prejeto od %s: %ld\r\n", xMessage.cSender,
66                      xMessage.lValue );
67              }
68              else
69              {
70                  UARTPrintf( "Neuspesen prejem sporocila.\r\n" );
71              }
72          }
73      }
74  }
75
76  int main ( void )
77  {
78      struct xProducerContext xProducer1Context, xProducer2Context;
79      xQueueHandle xQueue;
80
81      xQueue = xQueueCreate( 10, sizeof( struct xTaskMessage ) );
82
83      if( xQueue != NULL )
84      {
85          /* === Ustvari dva proizvajalca === */
86
87          /* Prvi proizvajalec naj posilja sporocila porabniku
88             ,,Consumer 2'' */
89
90          xProducer1Context.xQueue = xQueue;
91          strcpy( xProducer1Context.cReceiver, "Consumer 2" );
92
93          xTaskCreate( vProducer, "Producer 1", 512,
94              ( void * ) &xProducer1Context, 1, ( xTaskHandle * ) NULL );
95
96          /* Drugi proizvajalec naj posilja sporocila porabniku
```

```
97            ,,Consumer 1`` */
98
99          xProducer2Context.xQueue = xQueue;
100         strcpy( xProducer2Context.cReceiver, "Consumer 1" );
101
102         xTaskCreate( vProducer, "Producer 2", 512,
103             ( void * ) &xProducer2Context, 1, ( xTaskHandle * ) NULL );
104
105         /* === Ustvari dva porabnika === */
106
107         xTaskCreate( vConsumer, "Consumer 1", 512, ( void * ) xQueue, 2,
108             ( xTaskHandle * ) NULL );
109         xTaskCreate( vConsumer, "Consumer 2", 512, ( void * ) xQueue, 2,
110             ( xTaskHandle * ) NULL );
111
112         /* Zazeni razporejevalnik */
113
114         vTaskStartScheduler();
115     }
116
117     for( ; ; );
118 }
```

7.2 Izvedba sporočilne vrste

Izsek kode 7.2: Funkcije za delo s povezanim seznamom

```
1  struct ItemList_t
2  {
3      struct QueueItem_t xHead;
4      struct QueueItem_t xTail;
5      struct QueueItem_t *pxLast;
6  };
7
8  #define prvItemListIsEmpty( pxItemList ) \
9      ( pxItemList->xHead.pxNext == &pxItemList->xTail )
```

```
10
11 void itemListInit( struct ItemList_t *pxItemList )
12 {
13     pxItemList->xHead.pxNext = &pxItemList->xTail;
14     pxItemList->xTail.pxNext = NULL;
15     pxItemList->xHead.pucData = pxItemList->xTail.pucData = NULL;
16
17     pxItemList->pxLast = &pxItemList->xHead;
18 }
19
20 BaseType_t itemListIsEmpty( const struct ItemList_t *pxItemList )
21 {
22         return prvItemListIsEmpty( pxItemList );
23 }
24
25 struct QueueItem_t *itemListFirst( const struct ItemList_t *pxItemList )
26 {
27     struct QueueItem_t *pxResult;
28
29     if( !prvItemListIsEmpty( pxItemList ) )
30     {
31         pxResult = pxItemList->xHead.pxNext;
32     }
33     else
34     {
35         pxResult = NULL;
36     }
37     return pxResult;
38 }
39
40 void itemListPrepend( struct ItemList_t *pxItemList,
41                       struct QueueItem_t *pxItem )
42 {
43     if( prvItemListIsEmpty( pxItemList ) )
44     {
45         pxItemList->pxLast = pxItem;
46     }
47
```

```
48      pxItem->pxNext = pxItemList->xHead.pxNext;
49      pxItemList->xHead.pxNext = pxItem;
50  }
51
52  void itemListAppend( struct ItemList_t *pxItemList,
53                       struct QueueItem_t *pxItem )
54  {
55      pxItem->pxNext = &pxItemList->xTail;
56      pxItemList->pxLast->pxNext = pxItem;
57      pxItemList->pxLast = pxItem;
58  }
59
60  struct QueueItem_t *itemListRemove( struct ItemList_t *pxItemList )
61  {
62      struct QueueItem_t *pxResult;
63
64      pxResult = NULL;
65
66      if( !prvItemListIsEmpty( pxItemList ) )
67      {
68          pxResult = pxItemList->xHead.pxNext;
69          pxItemList->xHead.pxNext = pxResult->pxNext;
70          pxResult->pxNext = NULL;
71      }
72      return pxResult;
73  }
```

Izsek kode 7.3: Funkcija za pošiljanje sporočil

```
1  static BaseType_t
2  prvSendPrio( struct NamedQueue_t *pxNamedQueue, const void *pvMessage,
3               TickType_t xTicksToWait, enum ItemPriority_t xPrio,
4               BaseType_t xSendToBack )
5  {
6      BaseType_t xResult, xTimerRunning;
7      UBaseType_t uxQueueIndex;
8      xTimeOutType xTimeOut;
9
```

```
10      xResult = pdFALSE;
11      xTimerRunning = pdFALSE;
12      uxQueueIndex = pxNamedQueue->uxIndex;
13
14      do
15      {
16          xResult = xMutexTryLock( &( pxNamedQueue->xMutex ) );
17
18          if( xResult == pdTRUE )
19          {
20              xResult = !pxNamedQueueIndex->xEvent[ NQUEUE_CORE_ID
                    ].xPending;
21
22              if( xResult == pdTRUE )
23              {
24                  if( xSendToBack )
25                  {
26                      xResult = msgSendToBack( pxNamedQueue, pvMessage, xPrio
                            );
27                  }
28                  else
29                  {
30                      xResult = msgSendToFront( pxNamedQueue, pvMessage,
                            xPrio );
31                  }
32              }
33
34              if( xResult == pdTRUE )
35              {
36                  if( !listLIST_IS_EMPTY( &xTaskRecvList[ uxQueueIndex ] ) )
37                  {
38                      if( xTaskRemoveFromEventList(
39                          &xTaskRecvList[ uxQueueIndex ] ) )
40                      {
41                          portYIELD_WITHIN_API();
42                      }
43                  }
44
```

```
45                prvSendEvent( uxQueueIndex, SendEvent );
46            }
47
48            vMutexUnlock( &( pxNamedQueue->xMutex ) );
49        }
50
51        if( xResult == pdFALSE && xTicksToWait > 0 )
52        {
53            if( xTimerRunning == pdFALSE )
54            {
55                vTaskSetTimeOutState( &xTimeOut );
56                xTimerRunning = pdTRUE;
57            }
58
59            if( !xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) )
60            {
61                taskENTER_CRITICAL();
62                vTaskPlaceOnEventList( &xTaskSendList[ uxQueueIndex ],
63                                       xTicksToWait );
64                portYIELD_WITHIN_API();
65                taskEXIT_CRITICAL();
66            }
67            else
68            {
69                xTimerRunning = pdFALSE;
70            }
71        }
72    }
73    while( xResult == pdFALSE && xTimerRunning == pdTRUE );
74
75    return xResult;
76 }
```

Izsek kode 7.4: Funkcija za sprejemanje sporočil

```
1 static BaseType_t
2 prvReceive( struct NamedQueue_t *pxNamedQueue, void *pvMessage,
3             TickType_t xTicksToWait, BaseType_t xPeek )
```

```
4  {
5      BaseType_t xResult, xTimerRunning;
6      UBaseType_t uxQueueIndex;
7      void *pvResult;
8      xTimeOutType xTimeOut;
9
10     xResult = pdFALSE;
11     xTimerRunning = pdFALSE;
12     uxQueueIndex = pxNamedQueue->uxIndex;
13
14     do
15     {
16         xResult = xMutexTryLock( &( pxNamedQueue->xMutex ) );
17
18         if( xResult == pdTRUE )
19         {
20             xResult = !pxNamedQueueIndex->xEvent[ NQUEUE_CORE_ID
                   ].xPending;
21
22             if( xResult == pdTRUE )
23             {
24                 if( xPeek == pdFALSE )
25                 {
26                     pvResult = msgReceive( pxNamedQueue, pvMessage );
27                 }
28                 else
29                 {
30                     pvResult = msgPeek( pxNamedQueue, pvMessage);
31                 }
32                 xResult = ( pvResult != NULL );
33             }
34
35             if( xResult == pdTRUE )
36             {
37                 if( !listLIST_IS_EMPTY( &xTaskSendList[ uxQueueIndex ] ) )
38                 {
39                     if( xTaskRemoveFromEventList(
40                         &xTaskSendList[ uxQueueIndex ] ) )
```

```
41                  {
42                      portYIELD_WITHIN_API();
43                  }
44              }
45
46              prvSendEvent( uxQueueIndex, ReceiveEvent );
47          }
48
49          vMutexUnlock( &( pxNamedQueue->xMutex ) );
50      }
51
52      if( xResult == pdFALSE && xTicksToWait > 0 )
53      {
54          if( xTimerRunning == pdFALSE )
55          {
56              vTaskSetTimeOutState( &xTimeOut );
57              xTimerRunning = pdTRUE;
58          }
59
60          if( !xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) )
61          {
62              taskENTER_CRITICAL();
63              vTaskPlaceOnEventList( &xTaskRecvList[ uxQueueIndex ],
64                                     xTicksToWait );
65              portYIELD_WITHIN_API();
66              taskEXIT_CRITICAL();
67          }
68          else
69          {
70              xTimerRunning = pdFALSE;
71          }
72      }
73  }
74  while( xResult == pdFALSE && xTimerRunning == pdTRUE );
75
76  return xResult;
77 }
```

Izsek kode 7.5: Pošiljanje nasprotnega jedra o dogodku z uporabo prekinitev

```
1  static void
2  prvSendEvent( UBaseType_t uxIndex, enum EventType_t xEventType)
3  {
4      pxNamedQueueIndex->xEvent[ NQUEUE_CORE_ID ].uxIndex = uxIndex;
5      pxNamedQueueIndex->xEvent[ NQUEUE_CORE_ID ].xEventType = xEventType;
6      pxNamedQueueIndex->xEvent[ NQUEUE_CORE_ID ].xPending = pdTRUE;
7
8      __DSB();
9      __SEV();
10 }
```

Izsek kode 7.6: Prekinitveno servisna rutina za sporočilne vrste

```
1  void
2  vNamedQueue_IRQHandler( void )
3  {
4      struct NamedQueueEvent_t *pxEvent;
5      BaseType_t xDoYield;
6
7      pxEvent = &pxNamedQueueIndex->xEvent[ NQUEUE_PEER_CORE_ID ];
8      xDoYield = pdFALSE;
9
10     if( pxEvent->xEventType == SendEvent )
11     {
12         if( !listLIST_IS_EMPTY( &xTaskRecvList[ pxEvent->uxIndex ] ) )
13         {
14             xDoYield = xTaskRemoveFromEventList( &xTaskRecvList[
                   pxEvent->uxIndex ] );
15         }
16     }
17     else if( pxEvent->xEventType == ReceiveEvent )
18     {
19         if( !listLIST_IS_EMPTY( &xTaskSendList[ pxEvent->uxIndex ] ) )
20         {
21             xDoYield = xTaskRemoveFromEventList( &xTaskSendList[
                   pxEvent->uxIndex ] );
22         }
```

```
23      }
24
25      pxEvent->xPending = pdFALSE;
26
27      if( xDoYield )
28      {
29          portYIELD();
30      }
31  }
```

7.3 Primer uporabe

7.3.1 Glavno jedro

Izsek kode 7.7: Glavno opravilo temperaturnega krmilnika

```
1  static portTASK_FUNCTION( vTcTask, pvParameters )
2  {
3      struct TcConfig_t xLocalConfig;
4      uint16_t xSample, xAvgTemp;
5      uint32_t xRefreshCounter;
6      struct SignalItem_t xTempData;
7      portTickType xLastSampleTime;
8
9      xRefreshCounter = 0;
10     xLastSampleTime = xTaskGetTickCount();
11
12     while( pdTRUE )
13     {
14
15         taskENTER_CRITICAL();
16         memcpy( &xLocalConfig, &xTcConfig, sizeof( struct TcConfig_t ) );
17         taskEXIT_CRITICAL();
18
19         xSample = prvGetNewSample();
20         xAvgTemp = prvGetNewAverage( xSample );
```

```
21
22          if (xAvgTemp != UINT16_MAX)
23          {
24              xTempData.xTemperature = TO_CENTIGRADE(xAvgTemp);
25
26              if (xTempData.xTemperature / 10 >= xLocalConfig.xTempHi)
27              {
28                  xTempData.xPowerOn = xLocalConfig.xIsPositive;
29              }
30              else if (xTempData.xTemperature / 10 < xLocalConfig.xTempLo)
31              {
32                  xTempData.xPowerOn = !xLocalConfig.xIsPositive;
33              }
34
35              if ( xRefreshCounter == TC_REFRESH_PERIOD )
36              {
37                  xNamedQueueSendToFront( xSignalQueue, &xTempData, 0 );
38                  xRefreshCounter = 0;
39              }
40              else
41              {
42                  ++xRefreshCounter;
43              }
44
45              prvSwitchRelay( xTempData.xPowerOn );
46          }
47
48          vTaskDelayUntil( &xLastSampleTime, TC_SAMPLE_RATE );
49      }
50  }
```

Izsek kode 7.8: Opravilo za uveljavljanje nastavitev

```
1  static portTASK_FUNCTION( vTcCfg, pvParameters )
2  {
3      struct TcConfig_t xLocalTcConfig;
4      BaseType_t xResult;
5
```

```
6     while( pdTRUE )
7     {
8         xResult = xNamedQueueReceive( xConfigQueue, &xLocalTcConfig,
              portMAX_DELAY );
9         if( xResult == pdTRUE )
10        {
11            taskENTER_CRITICAL();
12            memcpy( &xTcConfig, &xLocalTcConfig, sizeof( struct TcConfig_t
                  ) );
13            taskEXIT_CRITICAL();
14        }
15
16        if( !xTcTaskStarted )
17        {
18            vTaskResume( xTcTask );
19            xTcTaskStarted = pdTRUE;
20        }
21    }
22 }
```

7.3.2 Pomožno jedro

Izsek kode 7.9: Opravilo za uporabniško konzolo

```
1  static portTASK_FUNCTION( vMOConsoleTask, pvParameters )
2  {
3      struct TcConfig_t xTcconfig;
4      NamedQueueHandle_t xConfigQueue;
5
6      ( void ) pvParameters;
7
8      xConfigQueue = xNamedQueueOpen( "tc_config", 1, sizeof( struct
           TcConfig_t ) );
9
10     memcpy(&xTconfig, &tc_default_config, sizeof(xTcConfig));
11
12     xDoConfigure = config_prompt();
```

```
13
14      if (xDoConfigure)
15      {
16          configure(&xTcConfig);
17      }
18      else
19      {
20          vUARTPrintf( TEST_UART, PRINT_LINE, "") ;
21          vUARTPrintf( TEST_UART, PRINT_LINE, "%s:", SETTINGS_TEXT );
22          config_print(&xTcConfig);
23      }
24
25      xNamedQueueSendToFront( xConfigQueue, &xTcConfig, 0 );
26
27      while( pdTRUE )
28      {
29          xDoConfigure = config_test();
30          if( xDoConfigure )
31          {
32              configure(&xTcConfig);
33              xNamedQueueSendToFront( xConfigQueue, &xTcConfig, 0 );
34          }
35      }
36  }
```

Izsek kode 7.10: Opravilo za prikaz stanja

```
1   static portTASK_FUNCTION( vM0InfoTask, pvParameters )
2   {
3       portTickType xFlashRate, xLastFlashTime;
4       NamedQueueHandle_t xSignalQueue;
5       struct SignalItem_t xSignalItem;
6       int xCurrentLed, xLedState;
7       BaseType_t xResult;
8
9       ( void ) pvParameters;
10
11      xFlashRate = ledFLASH_RATE_BASE + ( ledFLASH_RATE_BASE );
```

```
12      xFlashRate /= portTICK_RATE_MS;
13      xFlashRate /= ( portTickType ) 2;
14
15      led_set(RED_GPIO_PIN_BIT, 0);
16      led_set(GREEN_GPIO_PIN_BIT, 0);
17      led_set(BLUE_GPIO_PIN_BIT, 0);
18
19      xSignalQueue = xNamedQueueOpen( "tc_signal", 16, sizeof( struct
            SignalItem_t ) );
20
21      xNamedQueueReceive( xSignalQueue, &xSignalItem, portMAX_DELAY );
22
23      xCurrentLed = ( xSignalItem.xPowerOn ) ? GREEN_GPIO_PIN_BIT :
            RED_GPIO_PIN_BIT;
24
25      if( !xDoConfigure )
26      {
27          vUARTPrintf( TEST_UART, PRINT_LINE,
28                       "Temperatura ob zagonu: %d.%d st. Celzija\t\t",
29                       xSignalItem.xTemperature / 10,
30                       xSignalItem.xTemperature % 10 );
31      }
32
33      xLedState = 1;
34
35      xLastFlashTime = xTaskGetTickCount();
36
37      while( pdTRUE )
38      {
39          do
40          {
41              xResult = xNamedQueueReceive( xSignalQueue, &xSignalItem, 0 );
42          }
43          while( xResult != pdFALSE );
44
45          xCurrentLed = ( xSignalItem.xPowerOn ) ?
46                       GREEN_GPIO_PIN_BIT : RED_GPIO_PIN_BIT;
47
```

```
48          if( !xDoConfigure )
49          {
50              vUARTPrintf( TEST_UART, PRINT_RETURN,
51                          "Trenutna temperatura: %d.%d st. Celzija (%d)\t\t",
52                          xSignalItem.xTemperature / 10,
53                          xSignalItem.xTemperature % 10, xEventCount );
54          }
55
56          led_set( RED_GPIO_PIN_BIT, 0 );
57          led_set( GREEN_GPIO_PIN_BIT, 0 );
58          led_set( BLUE_GPIO_PIN_BIT, 0 );
59          led_set( xCurrentLed, xLedState );
60
61          vTaskDelayUntil( &xLastFlashTime, xFlashRate );
62
63          xLedState = !xLedState;
64      }
65  }
```

Slike

Literatura

[1] M. Weiser, ,,Some Computer Science Issues in Ubiquitous Computing“, *Communications of the ACM*, št. 36, zv. 7, str. 75 - 84, 1993

[2] D. S. Cannom, E. N. Prystowsky, ,,The Evolution of the Implantable Cardioverter Defibrillator, *PACE*, št. 27, str. 419 - 431, 2004

[3] (2015) Overview & History of ICD Therapy, Dostopno na: http://faculty.ksu.edu.sa/MFALREZ/EBooks Library/ECG and Cardiac/Overview ICD.pdf

[4] M. Bohr, ,,A 30 Year Retrospective on Dennard's MOSFET Scaling Paper“, *IEEE SSCS Newsletter*, str. 11 - 13, Winter 2007

[5] W. Stallings, *Computer Organization and Architecture Designing for Performance, 8th Edition*, Prentice Hall, 2010, str. 627-704

[6] B. Jacob, ,,Cache Design for Embedded Real-Time Systems“, *Embedded Systems Conference*, Danvers MA, ZDA, junij 1999

[7] (2015) Android, FreeRTOS top EE Times' 2013 embedded survey, Dostopno na: http://www.eetimes.com/document.asp?doc_id=1263083

[8] J. W. S. Liu, *Real-Time Systems*, Prentice Hall, 2000, str. 26-33

[9] (2014) Processes, Synchronization, and Deadlock. Dostopno na: http://6004.mit.edu/Spring14/handouts/L21-4up.pdf

[10] S. A. Ward, R. H. Halstead, *Computation Structures*, The MIT Press, 1999, str. 559-568

[11] *LPC43xx ARM Cortex-M4/M0 multi-core microcontroller user manual*, NXP B.V., 2014

[12] IEEE Std 1003.1, Single UNIX Specification Version 3, 2004 Edition

[13] H. Kopetz, *Real-Time Systems, Second Edition*, Springer, 2011, str. 38-40

[14] (2014) Message queue - Wikipedia. Dostopno na: http://en.wikipedia.org/wiki/Message_queue

[15] R. Barry, *Using the FreeRTOS Real Time Kernel, NXP LPC17xx Edition*, Real Time Engineers Ltd., 2011

[16] D. Kodek, *Arhitektura računalniških sistemov, 2. izdaja*, Bi-Tim, 2000, str. 47-68

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms, Second Edition*, The MIT Press, 2001, str. 162-164, 197-204

[18] *ARM Synchronization Primitives, Development Article*, ARM, 2009

[19] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts, 8th Edition*, John Wiley & Sons, 2009, str. 225-267

[20] G. L. Peterson, ,,Myths About the Mutual Exclusion Problem“, *Information Processing Letters*, št. 3, zv. 7, str. 115-116, 1981

[21] (2014) Embedded RTOS Queues, Dostopno na: http://www.freertos.org/Embedded-RTOS-Queues.html